

BASIC PROGRAMMING

HEATHKIT
CONTINUING
EDUCATION

Model EC-1100

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022
595-2045

Copyright © 1977
Heath Company
All Rights Reserved

Printed in the United States of America

CONTENTS

Foreword	8
What's All This About Computer Languages?	9
Introduction	11
Course Objectives	14

Part I

Tools of the Trade

SEGMENT ONE — CONVERSATIONS WITH A COMPUTER

Introduction	1-2
Unit Objectives	1-3
Programmed Instruction	1-4
Review Test	1-15
Review Answers	1-17

SEGMENT TWO — DECISIONS, DECISIONS!

Introduction	2-2
Unit Objectives	2-3
Programmed Instruction	2-4
Review Test	2-17
Review Answers	2-19

SEGMENT THREE — NUMBERS, PLEASE!

Introduction	3-2
Unit Objectives	3-3
Programmed Instruction	3-5
Review Test	3-21
Review Answers	3-23

SEGMENT FOUR — BUILD A DOGHOUSE!

Introduction	4-2
Unit Objectives	4-3
Let's Write a Program	4-4
Requirement 1	4-4
Requirement 2	4-5
Requirement 3	4-10
Make it Better	4-14
Formatting	4-14
Review Test	4-23
Review Answers	4-25

SEGMENT FIVE — HOW FUNCTIONS FUNCTION!

Introduction	5-2
Unit Objectives	5-3
Programmed Instruction	5-4
Review Test	5-29
Review Answers	5-31

SEGMENT SIX — LOOPY DE LOOP

Introduction	6-2
Unit Objectives	6-3
Programmed Instruction	6-4
Review Test	6-17
Review Answers	6-19

SEGMENT SEVEN — LISTS AND ARRAYS

Introduction	7-2
Unit Objectives	7-3
Programmed Instruction	7-4
Review Test	7-15
Review Answers	7-17

SEGMENT EIGHT — STRING SAVERS

Introduction	8-2
Unit Objectives	8-2
Programmed Instruction	8-4
Review Test	8-23
Review Answers	8-27

SEGMENT NINE — TRICKS OF THE TRADE

Introduction	9-2
Unit Objectives	9-3
Programmed Instruction	9-5
Review Test	9-25
Review Answers	9-29

Part II

Monument Building

Introduction to Part II	10-2
-------------------------------	------

SEGMENT TEN — HOW DO I START?

Introduction	10-4
What Shall We Do First?	10-5
Let's Calculate	10-6
What Should It Do?	10-6
Develop a Plan	10-9
Starting With The First Part	10-10
How Shall We Do It?	10-11
There Must Be A Better Way!	10-14
How About Vice-Versa?	10-18
What'll We Do For An Encore?	10-20
I'd Rather Do It Myself!	10-22
Put It All Together	10-23
Here's What We Have So Far	10-25
Here's Our Example Expression Again	10-26

Testing	10-30
Fancy Features	10-32
Test Your Skill At Splitting The Octals!	10-37
The Split Octal Sub-Program	10-38
The Complete Four-Base Calculator Program	10-40
Answers	10-42

SEGMENT ELEVEN — THE GRAND DESIGN

Introduction	11-2
Defining the Program	11-3
The Plan	11-4
Dividing the Program into Parts	11-5
Dealing the Cards	11-7
How to Represent the Cards	11-7
Naming the Cards and Suits	11-9
Card Names	11-11
Card Values	11-12
Assignment 1 — Define a Function	11-12
Position Numbers of Cards	11-12
Assignment 2 — A One-Dimensional Array for 52 Elements	11-13
Assignment 3 — Two-Dimensional Arrays (Cards & Suits)	11-13
Exercise A — Testing What You Have Done So Far	11-15
Dealing Cards (Cont'd)	11-17
Live, Dead, and In-Play Cards	11-17
Shuffling the Cards	11-18
Dealing Cards	11-18
Assignment 4 — Dealing	11-19
Transferring Cards From In-Play to Dead Group	11-20
Assignment 5 — Modifying the Shuffle	11-20
Exercise B — Testing the Card-Dealing Sub-Program	11-21
Answers	11-23
Hints	11-26

SEGMENT TWELVE — TAKE A CARD

Introduction	12-2
Playing Out Hands	12-3
The Player Information File	12-5
The Dealer's Hand	12-6

Players' Names	12-7
Assignment 1 — Names	12-7
Bankrolls and Bets	12-8
Assignment 2 — Bankrolls	12-8
Assignment 3 — Bets	12-8
A Test Printout	12-9
Playing Out Hands (Cont'd)	12-10
Assignment 5 — Playing the Hands	12-10
Assignment 6 — The Dealer's Upcard	12-11
Assignment 7 — Displaying Names and Suits	12-11
Scoring the Players Cards	12-12
Assignment 8 — The Score Computing Sub-Program	12-15
Assignment 9 — Scoring The First Two Cards ...	12-15
Assignment 10 — Check for Blackjack	12-15
Assignment 11 — Hit Me Dealer!	12-16
Zero Bets	12-18
Exercise A — The Program So Far	12-19
Answers	12-23

SEGMENT THIRTEEN — BEAT THE DEALER

Introduction	13-2
The Dealer Plays His Hand	13-3
Assignment 1 — Should the Dealer Play?	13-3
Assignment 2 — All Blackjacks or Bust	13-4
Assignment 3 — The Dealer Plays	13-4
Exercise A — Load All the Program So Far	13-7
Settling the Bets	13-9
Assignment 4 — Write a Program to Settle Bets .	13-11
Exercise B — Load the Settle Bets. Test the Program	13-12
Answers	13-13

SEGMENT FOURTEEN — GINGERBREAD AND BANDAGES

Introduction	14-2
Improvements	14-3
Assignment 1 — Dealer Checks for Blackjack	14-3
Testing the Program	14-4
Assignment 2 — Indicate When Cards are Shuffled	14-4
Assignment 3 — Indicate “Cards Being Dealt”	14-5
Assignment 4 — Indent Players’ & Dealer’s Hands	14-5
Assignment 5 — Separating the Hands	14-6
Assignment 6 — A “Settling Bets” Correction	14-7
Watson’s Test Game	14-8
Making Repairs	14-11
Assignment 7 — Positive-Value Bets	14-11
Assignment 8 — Limit Players’ Names to 10 Characters	14-11
Assignment 9 — Exclude Players With No Bankroll	14-11
Assignment 10 — Poll All Players’ Bankrolls	14-12
Program Completed	14-13
The Complete Blackjack Program	14-15
Answers	14-18

Part III

Appendixes

A. Description and Playing Rules for Blackjack	15-3
B. Microprocessors (Number Systems and Codes)	16-1
C. BENTON HARBOR BASIC	17-1
Index	18-1

FOREWORD

You are about to open the door to a new skill. We are happy to share the moment with you because there is nothing like the pride of accomplishment, joy, and satisfaction of learning.

It won't be a "piece of cake;" really worthwhile things seldom are. But we are going to do our best to make your work interesting, rewarding, and fun.

Since you will be providing your own motivation throughout the course, we are going to keep you busy so you won't get bored. There will be new information to absorb, questions to answer, exercises to do and, if you have a computer available, experiments to perform.

What can you expect when you have completed this course? Well, you will have learned how to use a set of tools whose purpose is to direct the operations of a computer. Much like carpenter's tools can be used to build anything from a doghouse to a cathedral, the programmer's tools can cause a computer to perform tasks that range from trivial to monumental. Again, like a carpenter, however, the knowledge of how the tools are used does not bestow the ability to build a cathedral; it is the imagination of the programmer, and his ability at abstract reasoning from which monuments arise.

Don't let this dissuade you! If your first product is a doghouse, so be it. Next, build a doghouse with two rooms and a second story. Then, progress to a whole kennel, and you're on your way to a monument. It is practice that makes perfect.

Beginning with the very first doghouse, you can expect delight in creating — and that is what makes programming fun. It has been said that a computer has the fascination of the ultimate pinball machine.

So deposit a coin and press the reset button — the first ball is in play.

WHAT'S ALL THIS ABOUT COMPUTER LANGUAGES?

Machine Language; Assembly Language; High-level Language; Interpreter; Compiler! Do these computer “buzz words” make any sense to you? Well then, maybe we can shed a little light on how people “talk” to computers and have them do their bidding.

The “heart” of a computer is a small part called the “Central Processing Unit”, or CPU for short. Beginning at some human-specified starting point in the computer’s memory, the CPU “looks” at the bits of information stored there. Let’s put ourselves in the position of the CPU and see what these “bits” of information look like.

Each location in a computer’s memory is known to the CPU by a number. The number is called that location’s “address,” just like the address of the place where you live. At each address, the CPU can “see” some electrical connections, some of which have a voltage on them and some of which do not. The number of connections at each address depends on the particular computer, but eight is a very common number in microcomputers, so we will use it in our examples.

Since there can be either a voltage or no voltage, each connection can assume only one of two conditions. With eight connections, the number of different combinations at a given address is 2^8 , or 256. The combinations range from no connection having a voltage to all connections having a voltage on them.

Each connection is called a “bit” of information and, in our example, the group of eight connections at an address is called a “byte”. When the CPU “sees” a particular combination of voltage/no voltage conditions on the eight bits at a byte location, it performs the function that its built-in “dictionary” says is associated with that combination. The particular combination, then, is an “instruction” to the CPU.

It is common notation to represent the voltage/no voltage conditions of the eight bits with "0" and "1" like this:

0 = No voltage

1 = Voltage

	B y t e							
Bit	7	6	5	4	3	2	1	0
	1	0	0	0	0	0	0	1

Or, simply:

10000001

To the popular 8080 microprocessor used in the CPU of many small computers, this instruction, 10000001, means "Add the value in my internal temporary storage Register named 'C' to the value in my internal arithmetic/storage Register named 'A'." That's quite a mouthful!

The instruction 10000001 is called "machine code" because it is the only instruction that the CPU "understands". That's such an important point that we want to repeat it: Each and every instruction that the CPU is capable of performing must be specified by a machine code such as 10000001.

Humans, on the other hand, find it tough to talk to a computer in a "language" that has words like "One,oh,oh,oh,oh,oh,oh,one", or, "Add the value in internal Register 'C' to the value in arithmetic Register 'A'." Since computers were developed to be the servant of man, man has taught the computer to "understand" languages that are easier for man to speak. "ADD C" would be easier for a human to remember than 10000001, and "ADD C" even gives a good hint at what function the CPU will perform when it sees the instruction. For these reasons, words like "ADD C" are called "mnemonics".

NOW HIRING — COMPUTER — MUST BE ABLE TO WORK HARD!

Almost any first-grader can add 2 plus 3 and come up with 5. Simple! Even this simple task is not all that easy for the CPU. Oh, sure, it can add 2 plus 3 with almost the speed of light, but look what it has to go through to do it:

MACHINE CODE INSTRUCTION	WHAT TO DO
00000110	Put the next byte of information into your Register 'B'
00000010	(This is the number '2' in machine language)
00001110	Put the next byte of information into your Register 'C'
00000011	(This is the number '3' in machine language)
01111000	Move the information in Register 'B' into your arithmetic Register 'A'
10000001	Add the value in Register 'C' to the value in Register 'A'
01010111	Move the answer into your Register 'D'

If we prepare a list of identical instructions using the more human-oriented mnemonic names for each operation, it's easier to understand what the CPU is doing at each step. A program of mnemonic instructions is said to be written in **Assembly Language**. Here's how our 2 + 3 program would look in Assembly Language:

```

MVI B, 2
MVI C, 3
MOV A,B
ADD C
MOV D,A

```

But in order to be able to write out the instructions in Assembly Language, the computer must have a program available that can "translate" the Assembly Language into the ones and zeros that the CPU can act upon. The program is called an "Assembler".

An Assembler reads each Assembly Language instruction and converts it to machine language, which it then stores somewhere in the computer's memory. The program written in Assembly Language is called "source code" and the ones and zeros that the CPU can use is called "object code". Here's our simple addition program:

<u>SOURCE CODE</u>	<u>OBJECT CODE PRODUCED BY THE ASSEMBLER</u>
MVI B, 2	00000110 00000010
MVI C, 3	00001110 00000011
MOV A,B	01111000
ADD C	10000001
MOV D,A	01011110

THERE'S GOT TO BE A BETTER WAY!

While the advent of Assembly Language made life easier for computer programmers, everyone yearned for an even more human-oriented way to express one's wishes to the computer. "How wonderful it would be," they cried, "if only we could simply tell the computer to make D equal 2 + 3." And so they did!

The result is called a "High-level Language". Our sample program can now be expressed in a very human-oriented form as:

D=2+3

What an improvement!

HOW DID HE DO THAT?

Generally speaking, High-level Languages are one of two types: Interpreters and Compilers. Here's how they work:

A Compiler reads the program you have written in High-level Language and produces object code for the CPU. After the Compiler has finished "compiling" your program into object code, it is no longer needed; the CPU can run the program directly from the object code produced by the Compiler.

The Compiler, then, is similar to an Assembler with the exception that it understands a more human-oriented and thus higher-level language.

An Interpreter works in much the same manner as someone who speaks Greek when you do not. If you wish to communicate with the Greek-speaking person, you must find an "interpreter" who speaks both Greek and the language that you understand, say English. You speak to the Interpreter in English and he speaks to the Greek in Greek. When the Greek speaks, the Interpreter repeats his words in English for you.

The BASIC programming language, about which this Course has been written, is most often implemented as an Interpreter form of computer language. You speak to it in English and it speaks to the CPU in machine code. In order for this to happen, you must have your Interpreter around any time you wish to speak to your computer in this High-level Language. In computer terminology, the Interpreter type of program is "resident" in your computer whenever you want to use its High-level Language.

The Interpreter knows "how to do" many different types of things. Each thing it knows how to do is a complete "fragment" program that exists in the memory space allocated to the Interpreter. For example, when you give the instruction `LET D=2+3` in BASIC, the Interpreter reads your instruction in English and branches off to a part of itself that contains the machine code:

```
00000110
00000010
00001110
00000011
01111000
10000001
01011110
```

When it has finished executing this small program, it returns to the command interpreter which reads your next English instruction, branches off to a part of the program that contains the machine code that does that assignment, and so on until the English program is finished.

Space in the computer's memory must be allocated for the Interpreter, in addition to space to store your English language program. The more "tricks" your Interpreter program can do, the larger the amount of memory that must be set aside to contain it.

The allocation of memory space to contain the various programs is called "memory mapping". The memory map of a typical microcomputer using the 8080 Integrated Circuit CPU might look like this (1 "K" of memory = 1024 bytes):

From 1 to 4 K	Special programs for operation of Computer's front panel and Input/Output capabilities
From 4 to 8 K	BASIC Interpreter program
The size of this area depends on how much memory has been installed in your computer	Space for the BASIC programs that you write
About 1/4 K	Special storage area used by BASIC
Last equipped memory location	Not equipped
Highest possible address = 64 K	

Here's a summary of computer languages:

LANGUAGE	WHAT IT DOES
Machine Assembly	The only language the CPU understands. CPU functions specified in human-oriented mnemonic form — produces machine language.
Compiler	Larger tasks specified in English — produces machine language.
Interpreter	Larger tasks specified in English — each task performed by a group of machine language instructions that are part of the Interpreter program.

INTRODUCTION

The workings of a computer are mysterious to most people. It is an interesting fact that an individual can instruct a computer to perform and still be quite ignorant of just how it actually does its magic. But then, many people can drive a car without having the slightest understanding of how an internal combustion engine works.

A set of instructions for a computer is called a program, and like the recipe for baking a cake, it is a series of steps which are to be performed in the order listed. There is a difference, though; the average would-be cake-baker would use the proper ingredient whether you spelled it sugar, suger, surgar or shugar. The computer follows your instructions slavishly; if you want sugar you must spell it exactly right because computers lack the ability to decide what you intended. They can only act upon what you actually said.

This is one of the most difficult parts of writing a computer program. The computer demands a preciseness for which humans are not generally noted. Let's take a moment to consider why this is so.

When computers were first invented, a group of highly-skilled people began to form. These people knew intimately how their machines worked deep inside the mass of vacuum tubes and wires.

You might compare these engineers to a skilled race car mechanic whose talent can make the engine run perfectly, but does not qualify him to race the car; the driver has the special skills needed on the track.

In the mid 1960's, John Kemeny and Thomas Kurtz, working at Dartmouth College, realized that some students had the ability to conceive marvelous tasks that computers would be able to perform, but were frustrated because they had not learned the special skills that would enable them to communicate their wishes to the computer. Kemeny and Kurtz reasoned that if a simple, easily-learned method of creating computer programs could be developed, many more people would be able to test and contribute their concepts. What was needed was a "language" for computer communication.

The new language was to have familiar English words which would cause the computer to perform standardized operations. When a number of these words were strung together to form a "program," the computer would perform each of the specified operations in turn to complete whatever task was desired. They called the language "Beginner's All-purpose Symbolic Instruction Code." It's known by the acronym BASIC, and today there is hardly a computer, big or small, which cannot be programmed with BASIC.

In order to make BASIC work, the computer has to have another program in operation that can interpret the words it reads and perform the operations desired. But the computer is really quite stupid. Fast, but stupid. If you tell it "GO TO" and it can't find that word in its list, then it does not know what you want it to do; even if the word "GOTO" is in its list.

And so, as programmers, we must not only choose the proper word to get the desired operation, but we must also give it in the correct "syntax." This means that it should be spelled correctly and have each necessary space or special character present and in the right place. Don't worry about what will happen if you get it wrong; the computer will just stop and tell you that you have made a "SYNTAX ERROR."

This course is divided into two main sections. The first section will describe all the standardized words of BASIC, and explain the operation that each word causes the computer to perform. These are the **tools** of BASIC language computer programming.

The second section of the course will describe the creative aspect of writing a computer program. Using the tools learned in the first section, you will participate in the decision factors that are part of writing a meaningful program, one that actually does something. You won't be building a cathedral, but it will be bigger than a doghouse and you should find it interesting.

The course material consists of text which will explain a part of BASIC in easy-to-grasp Segments. Sometimes we will use a technique called "programmed instruction." This type of instruction consists of a series of numbered paragraphs called "frames." Each frame gives a small amount of information, or asks a question about something you have already learned. Don't be concerned if the frames seem to be repetitious; it's important that you really understand exactly how the programming tools work and how they are used. The learning program repeats sometimes to lock the information firmly in your mind.

Each frame asks a question or two, and there are blanks provided for your answers. Don't just say the answers to yourself — write them down! This active participation goes a long way in helping you to remember.

A final note about the programmed instructions: The correct answers to the questions in a frame are given in a box immediately after the frame. That way, you will know immediately if you were right and can read back a few frames for review if you got it wrong. Use the yellow card to cover the frames below the one you are reading so you can't peek at the answer until you've written it down — you are on the honor system. Besides, you won't learn much if you cheat.

Try this example frame to see how the programmed instructions work:

0. The computer programming language called "Beginner's All-purpose Symbolic Instruction Code" is usually referred to by the acronym _____.

BASIC

That's how easy it is! Now, let's get started!

COURSE OBJECTIVES

When you have completed this course on BASIC Language Programming, you will be able to:

1. State what constitutes a computer "program".
2. Select the best description of the BASIC Programming Language.
3. Identify the name for a single, complete instruction in BASIC.
4. Name the individual parts of a BASIC Statement.
5. Describe a variable.
6. State the action taken by BASIC in response to each of a number of Statements.
7. Specify an item which must be used to construct a program.
8. Choose the action taken by BASIC if a Statement is typed without a line number.
9. State why line numbers are commonly assigned "by fives" or "by tens".
10. Describe BASIC's response to several examples of PRINT Statements.
11. Select the proper name for several types of arguments.
12. Choose the resultant PRINTout of several example programs.
13. Identify legal and illegal names for numeric and string variables.
14. Recognize the purpose of standard BASIC Statements by selecting the appropriate PRINTout produced by example programs which include the Statement.
15. Identify legally "nested" loops.
16. Choose the correct descriptions of numeric and string variables assigned in example programs.
17. State the number of elements, rows, and columns of several array variables mentioned in DIMension Statements.

Part I

Tools of the Trade



Individual Learning Program

BASIC PROGRAMMING

Segment 1

CONVERSATIONS WITH A COMPUTER

EC-1100

HEATH COMPANY
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977
Heath Company
All Rights Reserved
Printed in the United States of America

Segment 1

CONVERSATIONS WITH A COMPUTER

INTRODUCTION

Even though the BASIC programming language has been designed so it “understands” your native tongue (presuming you speak English), it has a limited vocabulary. To illustrate, BASIC knows what to do when you tell it “STOP”; but “HALT”, “WHOA”, “HOLD IT”, and other synonyms for “STOP” are all “Greek” to BASIC. This single fact is the reason for the existence of this course.

The goals of the first section of this course are to specify:

- Which of the English words that you know are also “understood” by BASIC.
- Exactly what BASIC does when you state one of the words that it “understands”.
- Exactly how you must state the words that BASIC “understands.”

Segment 1 describes some of the “ground rules” and introduces the words that will get you started in understanding how computer programs are written.

UNIT OBJECTIVES

When you have completed Segment 1, you will be able to:

1. Define the term "Computer Program".
2. Supply the name of single, complete instructions in BASIC.
3. Name the parts of a BASIC Statement.
4. Name the item that must begin a program line.
5. Provide the name for a string of characters enclosed by quotation marks.
6. Supply the name for an argument which is not enclosed by quotation marks.
7. Define what named storage locations in BASIC are called.
8. Select legal names for variables.
9. Explain what BASIC does when it reads the keyword LET in a program.
10. Define how BASIC obtains the value of an expression.
11. Specify the symbol for multiplication.
12. Specify the symbol for division.
13. Specify what a variable represents when it is used in an expression.
14. Explain how BASIC treats a portion of an expression which is enclosed in parentheses.

PROGRAMMED INSTRUCTION

1. A computer program is a list of **instructions** which the computer is to perform one after the other until it achieves your desired result. In this regard, a program is like a recipe for baking a cake, since it is a list of _____ that are to be performed in the order given.

instructions

2. In BASIC, a single complete instruction is called a "**Statement**". Here is an example of a single complete BASIC instruction:

STOP

The instruction "STOP", then, is a BASIC _____.

Statement

3. Most BASIC Statements are composed of two parts, a "**keyword**" and an "**argument**". The Statement "STOP", however, is complete in itself. That is, when the computer reads the instruction "STOP", it knows what to do and does not need any further information. "STOP" is a keyword. Thus, the Statement

STOP

does not require an _____.

argument

4. Here is another example of a BASIC Statement:

```
PRINT 2+2
```

In this Statement, the computer is being instructed to print something. What it is to print, "2 + 2", is the argument portion of the Statement and "PRINT" is the _____.

keyword

5. Look at the following two Statements.

```
PRINT 2+2  
STOP
```

"PRINT" and "STOP" are both _____, and
"2+2" is an _____.

keywords, argument

6. So that BASIC will know the order in which you wish the instructions performed, each line of your program **must** be given a "line number". You select the line numbers you wish at the time you write your program. BASIC will begin executing your instructions at the lowest line number you assign and, when it has finished with that line, it will execute the instruction at the next higher line number it can find.

It is common practice to assign line numbers "by fives" or "by tens" so there is room to add instructions if you find later that your program needs to be modified to work better.

In the program

```
10 PRINT 2+2  
20 STOP
```

the numbers 10 and 20 are called _____.

line numbers

Although all BASIC's are not the same in this regard, you will generally be safe if you use numbers in the range of 1 to 9999 as line numbers. Some computers allow numbers higher than 9999 to be used.

7. Let's review what you have learned so far:

- A. A complete instruction is called a _____.
- B. A Statement must begin with a _____.
- C. A Statement may require an _____.
- D. Each line of the program must begin with a _____.

A. Statement B. keyword C. argument D. line number

A typewriter-like keyboard, such as that shown in Photograph 1, is used to enter programs into the computer's memory. Each instruction line is typed on the keyboard until the entire program is typed. When the computer actually performs the list of instructions that comprise your program, it "reads" each Statement in its memory and performs the specified operation.

When the computer encounters a PRINT Statement, it presents some information to be seen or recorded. The information can be displayed on a television-like screen, called a CRT, such as the one shown in Photograph 2. If you want a permanent record of the information, you can connect a printer, such as the one shown in Photograph 3, to the computer. Whether a CRT or a printer is connected to the computer, the Statement PRINT is the one that causes messages and information to be "output". The combination of keyboard and CRT, or keyboard and printer, is called a terminal and is sometimes referred to as the system console.

Photograph 4 is a close-up view of a typical keyboard. We want to call your particular attention to a few of the keys.

Notice that there is a key in the upper left corner of the board that has a "1" on it. Unlike a typewriter, where a lower case letter "l" is sometimes used to represent the numeral "1", there is a special key on a computer keyboard for this number.

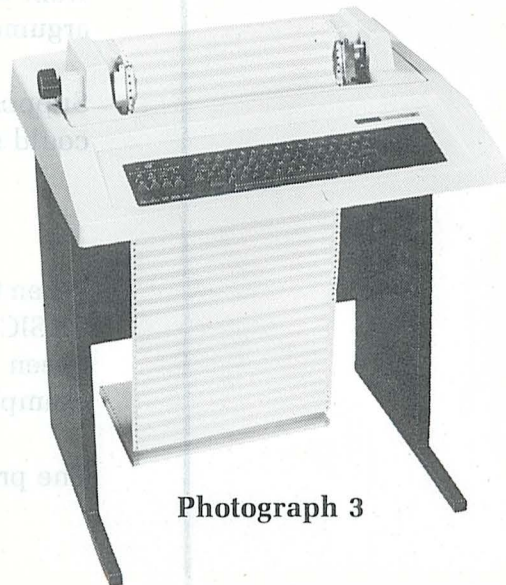
Notice also that there are separate keys for the letter "O" and the numeral "0".



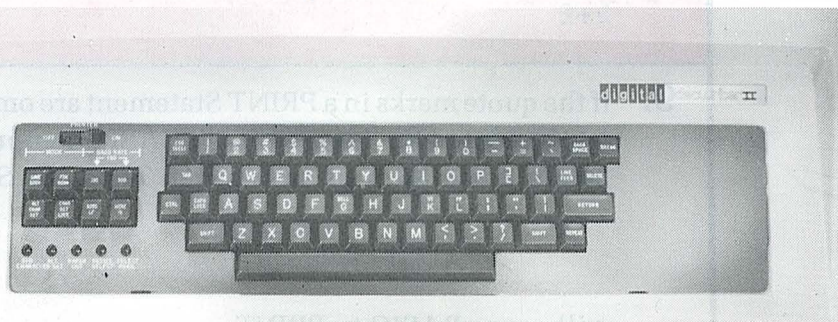
Photograph 1



Photograph 2



Photograph 3



Photograph 4

On the right of the keyboard, there is a key marked "RETURN". This key is pressed when you have finished typing a complete line of information and want the computer to store it in its memory or act on your command. (On some keyboards, the key is labeled "RET".)

Finally, over on the left you can see a key marked "CTRL" (Control). By holding this key down while you press another key, you can cause one of several special computer commands to be generated by the keyboard. For example, if the computer is doing something that you wish to interrupt, holding down the CTRL key and typing "C" will usually do it. When we want to refer to the act of holding CTRL while typing another key, we say, "Type CTRL/C", or CTRL/*plus whatever key is appropriate*.

8. Remember that PRINT is a keyword in BASIC, and unless you just want a blank line on your CRT or printer, you must furnish an argument to specify what is to be printed.

Suppose you wanted to PRINT your name. The program line could say

```
10 PRINT "WILLARD NICO"
```

When the argument is enclosed in quotation marks, as it is above, BASIC will **literally** PRINT the **string** of characters that are between the quotes. The argument, "WILLARD NICO", in the example above, is called a "**string literal**".

The program line

```
10 PRINT "2+2"
```

will cause BASIC to PRINT_____.

2+2

9. If the quote marks in a PRINT Statement are omitted, the argument is considered an "**expression**". BASIC **computes** all expressions **before** using them as arguments. Thus, the Statement

```
10 PRINT 2+2
```

will cause BASIC to PRINT _____.

10. Look at the following program:
- ```
10 PRINT"2+2="
20 PRINT 2+2
```

This will be PRINTed as:

2+2=  
4

In the above program, we say that

"2+2=" is a \_\_\_\_\_.  
2+2 is an \_\_\_\_\_.

string literal, expression

Turn to your Workbook and perform the exercises labeled "Exercise Group 1". Then return to this page.

There is more to learn about PRINT Statements, and we will be getting back to this later. Now, however, we want to go on to a different feature of BASIC so you can begin performing some experiments on your computer if one is available.

11. Imagine an office file cabinet filled with empty file folders and a group of alphabetical dividers. Suppose you take a sheet of paper and write the number 1976 on it, place the paper in the first folder behind the "A" alphabetical divider and write "A1" on the label tab of the file folder. What you have done is named a storage location (A1) and placed some information into storage (the number 1976).

In the same manner, you can name storage locations in BASIC and put information into storage. (For now, we will store only numbers.) Since the number stored in a particular location can be varied, and because everything has got to have a name, we call these storage locations "**variables**". Fortunately, you do not have to know where these storage locations are; BASIC can find them when you simply refer to the name you assign.

Named storage locations in BASIC are called \_\_\_\_\_.

variables



12. Some names are “legal” to use for variables, and some are not. A legal variable name **must** start with a letter and **may** have a single digit number after the letter. Two letters or two numbers are not permitted.

| <u>LEGAL<br/>VARIABLES</u> | <u>ILLEGAL<br/>VARIABLES</u> | <u>ILLEGAL BECAUSE</u> |
|----------------------------|------------------------------|------------------------|
| A                          | AA                           | Two letters            |
| Z1                         | 1Z                           | Starts with a number   |
| C2                         | C22                          | Two digit number       |
| B9                         | B9B                          | Letter after number    |

Thus, L4 is a \_\_\_\_\_ variable while 2L4 is \_\_\_\_\_.

legal, illegal

13. Here is a new BASIC Statement:

```
10 LET A=1976
```

In this Statement, LET is a keyword and A=1976 is an \_\_\_\_\_.

argument

14. In a LET Statement, the equal sign does not have the same meaning it does in algebra. (You don't have to remember its name, but if you are interested, it's called an “assignment operator”.) The equal sign in a LET Statement means “**compute** the expression to the right of the equal sign and place the **result** in the variable whose name is to the left of the equal sign”.

Thus, the Statement

```
10 LET A=1976
```

will cause variable A to have the value \_\_\_\_\_. (HINT: Since there are no + or – signs to the right of the equal sign, the expression 1976 is taken as a whole number that doesn't have to be computed.)

1976

15. If you use a variable name as part of an expression, you are telling BASIC to use the **content** of the named variable and not the name of the variable itself.

Thus, the program

```
10 LET A=1976
20 LET B=A
```

will cause variable B to have the value \_\_\_\_\_.

1976

16. Unless you assign a specific value to a variable, BASIC assumes that all variables contain the value zero. For example:

```
10 LET A=A+1
20 PRINT A
```

will PRINT

1

because variable A began with the value zero, and zero plus one equals one.

In the program

```
10 PRINT A
```

BASIC will PRINT\_\_\_\_\_.

0

It is very important for you to understand that the Statement

```
20 PRINT A
```

does not PRINT the letter A. What is PRINTed is the numeric value that is the **current content** of variable A.



17. Let's expand on that thought. An expression may consist of any mix of variable names, real numbers, and computations. Just remember that the computer will make any computations specified and use the **result** in the expression, and that the **content** of the variable is the value used when a variable is specified.

For example:

```
10 LET A=1976
20 LET B=A+2
30 PRINT B
```

In this program, BASIC will PRINT \_\_\_\_\_.

1978

18. How about:

```
10 LET A=1976
20 LET B=100
30 LET C=A+B
40 PRINT C
```

In this program, BASIC will PRINT \_\_\_\_\_.

2076

19. Unlike algebra, BASIC uses the symbol "\*" to indicate multiplication (so that you can use "X" for a variable name). For example:

```
10 LET A=2*2
20 PRINT A
```

Will cause BASIC to PRINT \_\_\_\_\_.

4

20. In BASIC, the symbol “/” means “divided by”. For example:

```
10 LET A=4/2
20 PRINT A
```

Will cause BASIC to PRINT\_\_\_\_\_.

2

21. Remember that a variable name may be used in an expression when you want the current **content** of the named variable used in making the computation.

What will the following programs PRINT?:

A. 10 LET A=1976  
20 PRINT A

A. \_\_\_\_\_

B. 10 LET A=1976+100  
20 PRINT A

B. \_\_\_\_\_

C. 10 LET A=1976  
20 LET B=50  
30 LET C=2\*B  
40 LET D=A+C  
50 PRINT D

C. \_\_\_\_\_

D. 10 LET A=1976  
20 LET B=100/2  
30 LET C=4\*B  
40 LET D=A-C  
50 PRINT D

D. \_\_\_\_\_

A. 1976    B. 2076    C. 2076    D. 1776



22. Just as in algebra, you can direct BASIC to make sub-computations and use the result in an expression. Enclosing the part you wish computed separately in parentheses gets the job done.

For example:

```
10 LET A=1976+(2*50)
20 PRINT A
```

will cause BASIC to PRINT:

2076

In the same way,

```
10 LET A=1976
20 LET B=100
30 LET C=A-(2*B)
40 PRINT C
```

will cause BASIC to PRINT \_\_\_\_\_.

1776

Turn to your Workbook and perform the exercises labeled "Exercise Group 2" (Page 4). Then return to this page.

In this first Segment of our course on the BASIC Programming Language, we have covered a number of the elementary features. In later Segments, these features will be used often in explaining other Statements and functions of BASIC, as indeed they will be used often in BASIC programs that you write.

Now see if you can answer the questions in the following test. This will do two things for you: First, it will let you see how much of the key information in Segment One you retained. And second, it will reinforce your learning and verify that you understand these key points. After you finish the questions, check your answers against those given in the "Review Answers". If you gave a wrong answer, go back and reread the proper frame, as listed in the "Frame for Review".

## REVIEW TEST

1. A computer program is a list of \_\_\_\_\_ for the computer to perform.
2. In BASIC, a single, complete instruction which the computer is to perform is called a \_\_\_\_\_.
3. Most (but not all) BASIC Statements consist of two parts: a \_\_\_\_\_ and an \_\_\_\_\_.
4. So that BASIC will know the order in which you wish your instructions performed, each line of the program must begin with a \_\_\_\_\_.
5. A string of characters enclosed by quotation marks is called a \_\_\_\_\_.
6. An argument which is not enclosed by quotation marks is called an \_\_\_\_\_.
7. Named storage locations in BASIC are called \_\_\_\_\_.
8. One of the following is not legal to use as a name for a variable. Which one?  
  - A. X
  - B. C1
  - C. 2B
  - D. A9

\_\_\_\_\_
9. Why is the name you selected in the previous question not legal to use as a name for a variable? \_\_\_\_\_
10. When BASIC reads the keyword LET in a program, what does it do?  

\_\_\_\_\_

\_\_\_\_\_

\_\_\_\_\_
11. An expression is \_\_\_\_\_ by BASIC to obtain its value.

12. An expression may consist of any mix of real numbers, computations, and \_\_\_\_\_.
13. The symbol for multiplication in BASIC is \_\_\_\_\_.
14. The symbol for division in BASIC is \_\_\_\_\_.
15. When a variable name is specified in an expression, BASIC uses the \_\_\_\_\_ of the named variable in computing the expression.
16. Enclosing part of an expression in parentheses instructs BASIC to \_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

## REVIEW ANSWERS

| <u>QUESTION</u> | <u>ANSWER</u>                                                                                                                                                                                    | <u>FRAME #<br/>FOR REVIEW</u> |
|-----------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| 1.              | Instructions. ....                                                                                                                                                                               | 1                             |
| 2.              | Statement. ....                                                                                                                                                                                  | 2                             |
| 3.              | Keyword, argument. ....                                                                                                                                                                          | 3                             |
| 4.              | Number of line number. ....                                                                                                                                                                      | 6                             |
| 5.              | String literal. ....                                                                                                                                                                             | 8                             |
| 6.              | Expression. ....                                                                                                                                                                                 | 9                             |
| 7.              | Variables. ....                                                                                                                                                                                  | 11                            |
| 8.              | C (2B). ....                                                                                                                                                                                     | 12                            |
| 9.              | Because it begins with a number. ....                                                                                                                                                            | 12                            |
| 10.             | (In your own words) Computes the value<br>of the expression to the right of the<br>equal sign and stores the result as<br>the value of the variable named to the<br>left of the equal sign. .... | 14                            |
| 11.             | Computed. ....                                                                                                                                                                                   | 17                            |
| 12.             | Variable names. ....                                                                                                                                                                             | 17                            |
| 13.             | * ....                                                                                                                                                                                           | 19                            |
| 14.             | / ....                                                                                                                                                                                           | 20                            |
| 15.             | Value or content. ....                                                                                                                                                                           | 21                            |
| 16.             | (In your own words) Perform the calcu-<br>lation enclosed in parentheses and use<br>the result in computing the rest of the<br>expression. ....                                                  | 22                            |







# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 2*

### DECISIONS, DECISIONS!

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## Segment 2

# DECISIONS, DECISIONS!

### INTRODUCTION

One of the things that sets a computer apart from a calculator is its ability to make decisions. If it didn't have this ability, a computer wouldn't be able to decide if a customer's charge account was "paid up" or "past due", and thus would not know whether to send a thank you note or a "pay up or else" warning. In this Segment, you will learn about some BASIC Statements that cause the computer to make decisions, and some other Statements which tell it what to do when it has come up with an answer.

## UNIT OBJECTIVES

When you have completed Segment 2, you will be able to:

1. Name one of the major things that distinguishes a computer from a calculator.
2. State the normal sequence of execution of a BASIC program.
3. Name a Statement which can be used to alter the normal sequence of execution of a BASIC program.
4. Describe the effect of the argument portion of a GOTO Statement.
5. State the decision made by BASIC when it encounters a Relational Statement.
6. Name the parts of a Relational Statement.
7. Match Relational Operators with their meanings.
8. Name the BASIC Statement Pair that tests Relational Statements.
9. Describe the way an IF . . . . THEN Statement functions.
10. Name the parts of an ON . . . . GOTO Statement.
11. Identify the integer portion of a number.
12. Describe the way an ON . . . . GOTO Statement functions.



## PROGRAMMED INSTRUCTION

1. Remember our BASIC program friends, "line numbers"? They are the first step in the process of decision-making. Normally, BASIC executes each line of your program beginning with the lowest line number, then the next higher number, and so on until it has executed the highest numbered program line. But you can instruct BASIC to alter this sequence whenever you wish by using a GOTO Statement!

When BASIC reads the keyword GOTO, it will jump to the program line specified in the argument and begin executing instructions from that point. Here's a simple example program to demonstrate how it works:

```
10 PRINT "2+2="
20 GOTO 40
30 PRINT "6"
40 PRINT "4"
50 STOP
```

When BASIC executes this program it will PRINT

```
2+2=
4
```

because the GOTO Statement in line 20 caused execution of the program to jump to line 40. Line 30, which was "skipped over", never got executed!

In line 20 of the example program, GOTO is a \_\_\_\_\_  
and 40 is an \_\_\_\_\_.

keyword, argument

2. The GOTO Statement can be used to jump backward in the program as well as forward. Here's a more complicated example:

```
10 PRINT "NOW IS THE TIME"
20 GOTO 50
30 PRINT "TO COME TO THE AID"
40 GOTO 70
50 PRINT "FOR ALL GOOD MEN"
60 GOTO 30
70 PRINT "OF THEIR COUNTRY."
80 STOP
```

BASIC will unscramble this message, and PRINT

```
NOW IS THE TIME
FOR ALL GOOD MEN
TO COME TO THE AID
OF THEIR COUNTRY.
```

The keyword GOTO causes BASIC to jump to the \_\_\_\_\_  
\_\_\_\_\_ specified in the argument and continue  
execution of the program from that point.

line number

3. Now that you know how to alter the usual order in which the numbered lines of a BASIC program are executed, it's time to consider the "relational tests" you can use to put the GOTO Statement to useful work.

If we told you that, numerically speaking, six was greater than five, you might say, "That's true!" What we did was state a relationship between two numbers and you evaluated the truth of what we stated. BASIC can do this, too! To see how this works, and what a powerful tool it is in writing programs, you need to learn a set of:

#### RELATIONAL OPERATORS

| <u>SYMBOL</u> | <u>MEANING</u>              |
|---------------|-----------------------------|
| =             | Is equal to                 |
| >             | Is greater than             |
| <             | Is less than                |
| >=            | Is greater than or equal to |
| <=            | Is less than or equal to    |
| <>            | Does not equal              |

For example, if you see

6>5

you read it as "six is greater than five". In this case, the answer is "That's True! Six is greater than five".

Remember, the important thing is: Is the Relational Statement true or false?

----- cont'd. -----



### 3. Cont'd.

Let's try a few:

| <u>RELATION</u> | <u>THAT'S TRUE</u> | <u>THAT'S FALSE</u> |
|-----------------|--------------------|---------------------|
| $6 > 5$         | X                  |                     |
| $5 > 6$         |                    | X                   |
| $5 = 5$         | X                  |                     |
| $6 < 5$         |                    | X                   |
| $2 * 2 = 4$     | X                  |                     |
| $25 > 100/4$    |                    | X                   |
| $25 > = 100/4$  | X                  |                     |
| $2 * 7 = 28/2$  | —                  | —                   |
| $5 < 6$         | —                  | —                   |
| $34/2 > 18$     | —                  | —                   |

True, True, False

## 4. A relational test consists of

An expression — A Relational Operator — An expression

Just like the expressions we discussed earlier, the expressions used in a relational test can consist of any mix of real numbers, variable names, and computations. Just remember that BASIC will make any computations specified and use the **result** in the expression, and that the **content** of the variable is the value used when a variable name is specified.

Suppose three variables are assigned as follows:

```
10 LET A=10
20 LET B=20
30 LET C=30
```

Then:

| <u>RELATION</u>        | <u>THAT'S TRUE</u> | <u>THAT'S FALSE</u> |
|------------------------|--------------------|---------------------|
| $2 * A = B$            | X                  |                     |
| $B > C$                |                    | X                   |
| $C = A + B$            | X                  |                     |
| $A = B - C$            |                    | X                   |
| $A = C - B$            | X                  |                     |
| $C / A = 2$            | _____              | _____               |
| $A * B = C$            | _____              | _____               |
| $(2 * A) + B = C + 10$ | _____              | _____               |

False, False, True

5. Now that you have GOTO and a powerful set of Relational Operators, you can put them to work with the great decision-maker IF . . . THEN.

Just as you might say to another person, “if you have enough money, **then** buy a soft drink”, we use the Statement Pair IF . . . THEN to instruct BASIC to make a decision and to tell it what we want done, depending on the result of the decision.

The keyword IF is always followed by a relational test argument. For example: IF  $2+2=4$ . BASIC decides if the outcome of the relational test is “true” or “false”. If the outcome is “false”, the rest of the program line is skipped and the program continues at the next higher numbered line. Only if the outcome of the relational test is “true”, will the Statements following the keyword THEN be executed.

Statements following the keyword THEN will be executed only if the outcome of the relational test is \_\_\_\_\_.

True

6. Let's see how this works:

```
10 PRINT "TWO PLUS TWO EQUALS"
20 IF $2+2=4$ THEN GOTO 50
30 PRINT "FIVE"
40 GOTO 60
50 PRINT "FOUR"
60 STOP
```

This program will PRINT

TWO PLUS TWO EQUALS  
FOUR

When BASIC evaluated the relational test “ $2+2=4$ ”, it found that relationship “true” and executed the Statement GOTO 50 which followed the keyword THEN. This caused the program to skip over the incorrect answer “FIVE”.

Statements following the keyword THEN will not be executed if the outcome of the relational test is \_\_\_\_\_.

False



7. Let's rearrange the program slightly:

```
10 PRINT "TWO PLUS TWO EQUALS"
20 IF 2+2=5 THEN GOTO 50
30 PRINT "FOUR"
40 GOTO 60
50 PRINT "FIVE"
60 STOP
```

BASIC will still PRINT

```
TWO PLUS TWO EQUALS
FOUR
```

Not because it is such a great mathematician, but because it found that the relational test " $2+2=5$ " was "false", and so it did not perform the Statement following THEN. Instead, it went directly to the next program line and PRINTed the correct answer.

Try this one:

```
10 IF 25<100 THEN GOTO 40
20 PRINT "25 IS GREATER THAN 100"
30 GOTO 50
40 PRINT "25 IS LESS THAN 100"
50 STOP
```

What will be PRINTed? \_\_\_\_\_

25 IS LESS THAN 100

8. Here are some examples of IF . . . . THEN Statements where **variable names** are used in the relational tests. Remember that the numerical content of a named variable will be used in evaluating the relational test:

```
10 LET A=150
20 LET B=75
30 IF A>B THEN GOTO 60
40 PRINT "A IS LESS THAN B"
50 GOTO 70
60 PRINT "A IS GREATER THAN B"
70 STOP
```

This program will PRINT

A IS GREATER THAN B

How about:

```
10 LET A=150
20 LET B=75
30 IF A=2*B THEN GOTO 60
40 PRINT "A DOES NOT EQUAL TWICE B"
50 GOTO 70
60 PRINT "A EQUALS TWICE B"
70 STOP
```

This program will PRINT: \_\_\_\_\_

A EQUALS TWICE B

**9. Quick quiz:**

The keyword GOTO causes BASIC to jump to a specified \_\_\_\_\_ and continue execution of the program from that point.

The line number that BASIC is to jump to must be specified as the \_\_\_\_\_ portion of the GOTO Statement.

The symbols >, >=, <> and < are all \_\_\_\_\_ operators.

36>25 is a relational \_\_\_\_\_.

line number, argument, relational, expression (or test)

**10. More quiz:**

In an IF...THEN Statement, the instruction following the keyword THEN will only be executed if the relational expression evaluates as (true) (false).

If the relational expression in an IF...THEN Statement evaluates as false, the instruction following the keyword THEN will be ignored and execution of the program will continue at the next \_\_\_\_\_ line number.

A relational expression in an IF...THEN Statement can include any mix or real numbers, computations, and \_\_\_\_\_.

When a variable name is included in a relational expression, or any other expression, it is the \_\_\_\_\_ of the variable that is used in computing the expression.

true, higher, variable names, content

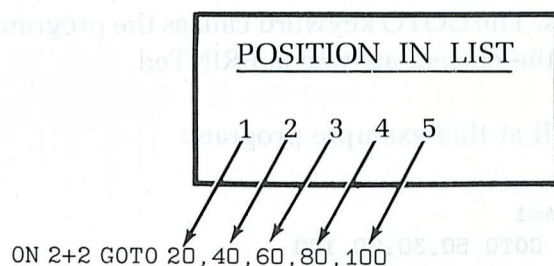
Turn to your Workbook and perform the exercises labeled "Exercise Group 3". Then return to this page.



11. Since decision-making is the most important capability of a computer, BASIC includes several Statements to make it easy for the programmer to call for a decision. You have already learned about the relational tests that are used with the IF . . . THEN Statement. Another useful programming tool, the ON . . . GOTO Statement, provides a way to have BASIC select from a **list** of line numbers when making a jump in program execution.

When BASIC reads the keyword ON, it computes the argument which immediately follows the keyword in the same manner as the other expressions we have discussed. Then it calculates the integer of that computation (the integer is the part that remains after any decimal portion is discarded). The value of the integer specifies the **position** in the list of line numbers following the keyword GOTO. Program execution then jumps to the line number in the specified position.

An ON . . . GOTO Statement is constructed like this:



The number 80 at position 4 in the list refers to a

line number

**12.** Here's an example of how ON . . . GOTO might be used:

```
10 ON 2+2 GOTO 20,40,60,80,100
20 PRINT "2+2=1"
30 GOTO 110
40 PRINT "2+2=2"
50 GOTO 110
60 PRINT "2+2=3"
70 GOTO 110
80 PRINT "2+2=4"
90 GOTO 110
100 PRINT "2+2=5"
110 STOP
```

This program will cause BASIC to PRINT

2+2=4

because the expression "2+2" following the keyword ON computes to 4, and line number 80 is in the fourth position in the list of line numbers. The GOTO keyword causes the program to jump to line 80 and the correct answer is PRINTed.

Try your skill at this example program:

```
10 LET A=1
20 ON A GOTO 50,30,70,100
30 PRINT "ARE HARDLY"
40 GOTO 80
50 PRINT "HEDGEHOGS"
60 GOTO 80
70 PRINT "HANDSOME"
80 LET A=A+1
90 GOTO 20
100 STOP
```

----- cont'd. -----

## 12. Cont'd.

Tricky - maybe so, but with what you have learned about BASIC so far, you should be able to figure out that this program will PRINT:

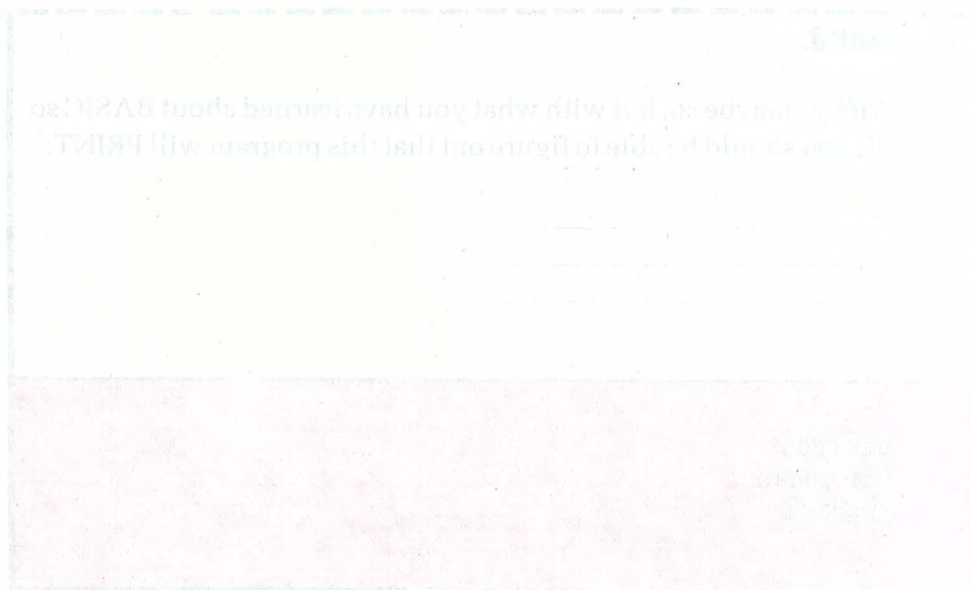
\_\_\_\_\_  
\_\_\_\_\_  
\_\_\_\_\_

HEDGEHOGS  
ARE HARDLY  
HANDSOME

It's bound to happen sometime, so it would be wise for you to understand what will happen if the argument in an ON . . . . GOTO Statement computes to either 0 (zero), or to a number that is higher than the last position in your list of line numbers.

In either of the above cases, the entire ON. . . .GOTO Statement will be ignored and program execution will continue at the next higher line number after the ON. . . .GOTO Statement.





and to help you understand me, so it would be wise for you to understand  
what will happen if the argument is an ON... GOTU Statement  
that starts to either a (zero), or to a number that is higher than the last  
number in your list of line numbers.

... of the above cases, the entire ON... GOTU Statement will be  
... and program execution will continue at the next higher line  
... after the ON... GOTU Statement.

## REVIEW TEST

1. One of the major things that distinguishes a computer from a calculator is its ability to make \_\_\_\_\_.
2. Normally, BASIC executes programs beginning with the lowest line number and continuing with the next \_\_\_\_\_ line number in sequence.
3. The BASIC Statement that can alter the sequence of execution of line numbers is \_\_\_\_\_.
4. The argument portion of a GOTO Statement specifies the \_\_\_\_\_ to which the program is to jump and continue execution.
5. When BASIC encounters a Relational Statement, it decides if the Statement is \_\_\_\_\_ or \_\_\_\_\_.

6. A relational test consists of:

An Expression — A \_\_\_\_\_ — An Expression

7. List the Relational Operator Symbols for each of the following meanings:

Meaning

Symbol

Is equal to

\_\_\_\_\_

Is greater than

\_\_\_\_\_

Is less than

\_\_\_\_\_

Is greater than or equal to

\_\_\_\_\_

Is less than or equal to

\_\_\_\_\_

Does not equal

\_\_\_\_\_

8. The BASIC Statement Pair that tests Relational Statements is \_\_\_\_\_.
9. In an IF...THEN Statement, the Statement following the keyword THEN will only be executed if the Relational Statement is \_\_\_\_\_.
10. In an IF...THEN Statement, if the Relational Statement is false, the Statement following the keyword THEN is ignored and program execution continues at the \_\_\_\_\_.
11. The Expressions in a Relational Statement can include any mix of real numbers, computations, and \_\_\_\_\_.
12. In an ON...GOTO Statement, an \_\_\_\_\_ must follow the keyword ON.
13. In an ON...GOTO Statement, a \_\_\_\_\_ of line numbers must follow the keyword GOTO.
14. In an ON...GOTO Statement, each line number can be specified by its \_\_\_\_\_ in the list of line numbers.
15. The integer of the number 3.1416 is \_\_\_\_\_.
16. When the argument in an ON...GOTO Statement is computed, the \_\_\_\_\_ of the result selects the position of the line number in the list that follows the keyword GOTO.
17. When an ON...GOTO Statement is executed, the program jumps to the line number which was in the \_\_\_\_\_ of the line number list that was computed by the argument following the ON keyword.
18. If the computed result of an ON...GOTO Statement equals zero, or is higher than the last position in the list of line numbers, execution of the program will \_\_\_\_\_.



## REVIEW ANSWERS

| QUESTION | ANSWER                            | FRAME #<br>FOR REVIEW |
|----------|-----------------------------------|-----------------------|
| 1.       | Decisions. ....                   | Introduction          |
| 2.       | Higher. ....                      | 1                     |
| 3.       | GOTO. ....                        | 1 & 2                 |
| 4.       | Line number. ....                 | 1 & 2                 |
| 5.       | True or false. ....               | 3 & 4                 |
| 6.       | Relational Operator. ....         | 4                     |
| 7.       | Meaning                           | Symbol                |
|          | Is equal to .....                 | =                     |
|          | Is greater than .....             | >                     |
|          | Is less than .....                | <                     |
|          | Is greater than or equal to ..... | >=                    |
|          | Is less than or equal to .....    | <=                    |
|          | Does not equal .....              | <>                    |

|     |                                                                                            |    |
|-----|--------------------------------------------------------------------------------------------|----|
| 8.  | IF....THEN .....                                                                           | 5  |
| 9.  | True.....                                                                                  | 5  |
| 10. | Next higher line number. ....                                                              | 6  |
| 11. | Variable names. ....                                                                       | 8  |
| 12. | Argument.....                                                                              | 11 |
| 13. | List. ....                                                                                 | 11 |
| 14. | Position. ....                                                                             | 11 |
| 15. | 3.....                                                                                     | 11 |
| 16. | Integer. ....                                                                              | 11 |
| 17. | Position. ....                                                                             | 11 |
| 18. | Continue at the next higher line number<br>after the ON....GOTO Statement. .... Conclusion |    |

**HEATHKIT  
CONTINUING  
EDUCATION**

# **Individual Learning Program**

## **BASIC PROGRAMMING**

*Segment 3*

**NUMBERS, PLEASE!**

**EC-1100**

**HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022**

**Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America**



## Segment 3

# NUMBERS, PLEASE!

### INTRODUCTION

While the computer is capable of doing grand things, it would be of little use if there were no way to feed it numbers and other information (called **data**) to process. In this Segment, we will explore some of the ways your BASIC program can be provided with numerical data.

First, though, let's revisit the PRINT Statement which was discussed in Segment 1. Two simple additions to what you have learned about PRINT will make it possible to format lines of printing for better appearance and readability.

## UNIT OBJECTIVES

When you have completed Segment 3, you will be able to:

1. Describe how a single PRINT Statement can cause several data items to be PRINTed on the same line.
2. Describe how several data items can be PRINTed on the same line and spaced to align into columns.
3. Name the two special characters that are used to separate multiple data items following a PRINT Statement and state how each is used.
4. State the number of characters in a PRINTed column.
5. Explain how data items from several PRINT Statements can be PRINTed on the same line.
6. Name the Statement Pair that can instruct BASIC to assign sequential data values from a list.
7. Explain the reason for using a "stopper" as the last value in a DATA list.
8. State what a READ Statement has as its argument.
9. Explain how data items from several DATA Statements are considered by BASIC.
10. Explain how data is READ from the DATA list.
11. Name the special character that separates variables in the argument of a READ Statement.
12. Describe what occurs if too few data items are provided in DATA Statements.
13. Name the Statement that allows comments, headings, and explanatory text to be added to BASIC programs.
14. Describe how the computer responds when BASIC encounters a REMark Statement.

15. Name the Statement that allows data to be provided to a program from the Console keyboard.
16. Name the special character that BASIC prints as a prompt when data is required by an INPUT Statement.
17. Name the special character that the keyboard operator must use to separate data items typed on the keyboard when several data items are required by an INPUT Statement.
18. State what occurs if less data items are typed on the keyboard than are required by an INPUT Statement.
19. State what occurs if more data items are typed on the keyboard than are required by an INPUT Statement.
20. Name the item that can be substituted for the question mark prompt printed by BASIC when it encounters an INPUT Statement.
21. Explain how to incorporate a prompt message as part of an INPUT Statement.



## PROGRAMMED INSTRUCTION

1. Up to now, we have been using separate PRINT Statements for each data item. Because of this, BASIC has PRINTed each item on a separate line.

For example:

```
10 PRINT "2+2="
20 PRINT 2+2
30 STOP
```

will PRINT

```
2+2=
4
```

Notice that the string literal "2+2=" was PRINTed exactly as called for in line 10. The computed value, 2+2, was PRINTed with a space in front of the number. The reason for the space is that BASIC reserved a place to put a minus sign had the value been a negative number.

Suppose we wanted to format the PRINTed output so that it would appear as follows:

```
2+2= 4
```

Ever-accommodating BASIC will do this if we arrange the items to be PRINTed in a **list**.

```
10 PRINT "2+2=";2+2
20 STOP
```

will PRINT

```
2+2= 4
```

Following a PRINT Statement, an argument with two or more data items is called a \_\_\_\_\_.

list

2. In the example given in Frame 1, the data items in the list following the PRINT Statement are separated by a semicolon. The semicolon tells BASIC not to add any spaces between the items when PRINTing them. (The space reserved for the minus sign will always appear as part of a computed number if the number is positive.)

Here's an example with several items in a list to be PRINTed:

```
10 PRINT 2+2;3+3;4+4;5+5;6+6
20 STOP
```

and what BASIC will PRINT:

4 6 8 10 12

A \_\_\_\_\_ separating items in a list to be PRINTed tells BASIC not to add spaces between the items.

semicolon

3. The semicolon can also be used to join a number of PRINT Statements on separate program lines into a single list. In effect, the semicolon tells BASIC not to start a new line when the next PRINT Statement is encountered, even if it is several program lines away.

For example:

```
10 PRINT "2+2=";
20 PRINT 2+2
30 STOP
```

will PRINT

2+2= 4

When a PRINT Statement ends with a semicolon, data from the next PRINT Statement will be PRINTed on the \_\_\_\_\_ line.

same

4. The semicolon works for PRINTing string literals as well as numbers, or a mixture of both. Just remember that BASIC will not add any spaces. If you join two or more string literals, you will have to consider the need for a space between words and include one in the string if indicated.

For example:

```
10 PRINT "NOW IS THE TIME ";
20 PRINT "FOR ALL GOOD MEN ";
30 PRINT "TO COME TO THE AID ";
40 PRINT "OF THEIR COUNTRY"
50 STOP
```

will PRINT

NOW IS THE TIME FOR ALL GOOD MEN TO COME TO THE AID OF THEIR COUNTRY

A list of string literals, variable contents, and computed numbers can be constructed as the argument of a PRINT Statement. The semicolon separates items in the list and tells BASIC to PRINT them without adding any spaces.

For example:

```
10 LET A=5
20 LET B=6
30 PRINT A;"TIMES";B;"EQUALS";A*B
40 STOP
```

will PRINT

5 TIMES 6 EQUALS 30

BASIC will PRINT a mixed list of string literals, variable contents, and computations without adding any spaces if the items are separated by a \_\_\_\_\_.

semicolon



5. Commas may be used in place of semicolons to separate items in a list to be PRINTed. The difference is that commas tell BASIC to add the number of spaces required to align the PRINTed data items into **columns**.

BASIC considers a line of printing to be divided into columns, each 14 characters wide. When BASIC encounters a comma while PRINTing a list of data items, it automatically spaces over to the beginning of the next column.

Here's how it works:

```
10 PRINT "APPLES", "ORANGES", "PEARS", "BANANAS"
20 PRINT "PEACHES", "PLUMS", "WATERMELONS", "KUMQUATS"
30 STOP
```

and the result:

|         |         |             |          |
|---------|---------|-------------|----------|
| APPLES  | ORANGES | PEARS       | BANANAS  |
| PEACHES | PLUMS   | WATERMELONS | KUMQUATS |

The comma can be used to "columnize" data in PRINT Statements that are on different program lines.

```
10 PRINT "APPLES",
20 PRINT "ORANGES",
30 PRINT "PEARS",
40 PRINT "BANANAS"
50 STOP
```

This program will PRINT

|        |         |       |         |
|--------|---------|-------|---------|
| APPLES | ORANGES | PEARS | BANANAS |
|--------|---------|-------|---------|

The comma can be used to separate data to be PRINTed into \_\_\_\_\_, each 14 characters wide.

columns

6. Now, let's consider some of the ways that you can provide BASIC with some data to feed on when running a program. So far, the only way of providing data that we have considered is the direct assignment of a variable. That is:

```
10 LET A=50
20 STOP
```

While this method is useful in many cases, it is cumbersome when several values are to be processed. By using the READ. . . DATA Statement pair, BASIC can be instructed to assign sequential data values from a list.

Here is an example:

```
10 READ A
20 IF A=999 THEN GOTO 60
30 PRINT 2*A;
40 GOTO 10
50 DATA 1,2,3,4,5,6,7,8,9,10,999
60 STOP
```

This program will PRINT

2 4 6 8 10 12 14 16 18 20

Let's analyze how the program functions. Line 10 tells BASIC to READ a data value into variable A. BASIC scans through the program until it finds the first DATA Statement, in this case at line 50. The first value in the DATA list is assigned as the value of variable A. (Note that items in the DATA list are separated by commas.)

The content of variable A is tested for 999 by the IF. . . THEN Statement in line 20. The value 999 has been added as the last value in the DATA list as a "stopper"; an identifiable number to tell us when all the data has been used.

----- cont'd. -----

**6. Cont'd.**

Line 30 tells BASIC to PRINT the content of the A variable multiplied by two. (Note the use of the semicolon to call for all results to be PRINTed on the same line.) Line 40 sends the program back to line 10 for another round. Again, line 10 instructs BASIC to READ a value from the DATA list. BASIC remembers that the last value it read was in position one, so this time, it READs the value from position two into the A variable.

The cycle repeats until the "stopper" is READ into the A variable. Now the Relational test in line 20 comes up "true" and causes the program to jump to the STOP Statement in line 60. The STOP Statement terminates the program.

A "stopper" is added as the last item in a DATA list so your program can make a test to determine \_\_\_\_\_

when all DATA items have been used.



7. There can be several READ Statements in a program, and there can be several DATA Statements, too. BASIC considers that **all** DATA Statements comprise a **single list** of data. When all items in the first DATA Statement are used, data will come from the second DATA Statement the next time a READ Statement is encountered.

This example shows how it works:

```
10 READ A,B
20 IF B=999 THEN GOTO 70
30 PRINT A*B;
40 GOTO 10
50 DATA 5,6,2,2,4,9,2,8
60 DATA 7,7,3,6,4,7,999,999
70 STOP
```

This program will PRINT

```
30 4 36 16 49 18 28
```

Program line 10 tells BASIC to READ two items from the DATA list; one into variable A and the second into variable B. The next time line 10 is executed, the data READ into variables A and B will be the third and fourth items in the DATA list, and so on.

The semicolon at the end of line 30 tells BASIC to PRINT all results of the "A\*B" computation \_\_\_\_\_.

(in your own words) on the same line, without spaces.

8. Better understanding of how to use a tool comes from knowing what will happen if the tool is misused; which brings up the question of what happens if your program tries to READ more data than you have provided in the DATA Statements.

Consider this:

```
10 READ A
20 PRINT A;
30 GOTO 10
40 DATA 1,2,3,4,5
50 STOP
```

This program will loop through the READ, PRINT and GOTO lines five times. When it tries to execute the loop the sixth time, it finds that there are no more data items left in the DATA Statement and will print an error message telling you

```
1 2 3 4 5
! ERROR - NO DATA AT LINE 10
```

Look carefully at the following.

```
40 READ A,B
50 PRINT A*B,
60 GOTO 40
70 DATA 1,2,3,4,5,6,7
80 STOP
```

What will be PRINTed?

```
2 12 30
! ERROR - NO DATA AT LINE 40
```

As we have mentioned, there are some minor variations between different versions of BASIC; this is particularly true of the exact form of the error messages. For all of our examples, we have used the Benton Harbor version of BASIC. The error messages described in Frame 8 might appear in a different version of BASIC, such as

OUT OF DATA IN LINE 10

But whatever form the message takes, its meaning will be clear.

9. Including a "stopper" in your DATA list, and testing for it as shown in the example programs of Frames 6 and 7, is one way to determine when the program should end and avoid the error message. Because we were READing values into two variables in Frame 7, we included two "stoppers" in the DATA Statement. If we had put only one "stopper" in the list, it would have been read into the A variable and, since there would have been no more data for the B variable, an error message would have resulted.

When using a "stopper" to signal the end of a DATA list, it must be placed so that it will be READ into the \_\_\_\_\_ variable in the READ Statement.

last



10. Here's an interesting diversion in our discussion of feeding your programs data to process.

When you write a computer program in BASIC, or any other language, a time will come when you look at what you have written and wonder, "What does this part of the program accomplish", or "Why in the world did I include these Statements". Surely, someone else who reads your program will wonder just what you are trying to do and how the program is supposed to work. A way out of this dilemma is to "**document**" the program by including remarks that tell what the program is supposed to accomplish and how the various parts work.

BASIC includes a REMark Statement which is completely ignored by the computer when the program is executed, yet allows you to add explanatory text as part of the program.

For example:

```
10 REM THIS PROGRAM MULTIPLIES TWO NUMBERS
20 REM FIRST, READ THE NUMBERS
30 READ A, B
40 REM THEN MULTIPLY THEM TOGETHER
50 LET C=A*B
60 REM AND PRINT THE RESULT
70 PRINT C
80 REM HERE IS WHERE WE GET OUR DATA
90 DATA 6, 8
100 REM THE NEXT STATEMENT ENDS THE PROGRAM
110 STOP
```

This program will PRINT

48

The important thing to notice is that when BASIC reads the REMark Statement, it **ignores** everything else on that program line.

REMark Statements can be used to \_\_\_\_\_  
\_\_\_\_\_ your BASIC programs.

document or explain

Let's write a score averaging program to practice your knowledge about BASIC and put some of the Statements you have learned so far to practical use. To compute an average score, all scores are totaled and the total is divided by the number of scores in the list. Here's a program with REMark comments to compute an average score. (For emphasis, we use asterisks in REMark lines containing the title of the program. The blank REMark line isolates the title.)

```
10 REM ** THIS PROGRAM COMPUTES THE AVERAGE OF
20 REM ** A NUMBER OF SCORES.
30 REM
40 REM READ A SCORE INTO VARIABLE 'S'
50 READ S
60 REM CHECK FOR 'STOPPER'
70 IF S=999 THEN GOTO 170
80 REM ADD SCORE TO TOTAL IN VARIABLE 'T'
90 LET T=T+S
100 REM COUNT NUMBER OF SCORES READ
110 LET N=N+1
120 REM LOOP FOR NEXT SCORE
130 GOTO 50
140 REM HERE IS THE LIST OF SCORES
150 DATA 78,82,94,87,84,86,67,92,88,999
160 REM PRINT RESULTS OF COMPUTATION
170 PRINT "THE AVERAGE OF";N;"SCORES IS";T/N
180 STOP
```

The Score Averaging program illustrates several of the facts you have learned about BASIC:

| <u>LINE NUMBER</u> | <u>FACT ILLUSTRATED</u>                                                                     |
|--------------------|---------------------------------------------------------------------------------------------|
| 10                 | REMark Statement used to describe what program does                                         |
| 40                 | REMark Statement used to explain operation                                                  |
| 50                 | Data READ into variable from DATA list                                                      |
| 70                 | "Stopper" used to mark end of data accumulation                                             |
| 90                 | Use of a variable to keep running total                                                     |
| 110                | Use of a variable to count items                                                            |
| 130                | Use of a GOTO Statement to alter program sequence                                           |
| 150                | Multiple values in a DATA list                                                              |
| 170                | Use of semicolon to PRINT string literal, variable content and computed result on same line |

When executed on a computer, the Score Averaging program will PRINT

THE AVERAGE OF 9 SCORES IS 84.2222



11. Still another method of obtaining data for your BASIC program to process is by **direct entry** from the Console keyboard.

When BASIC encounters the Statement INPUT, it waits for data to be manually typed on the Console keyboard. To "prompt" the operator that data is to be typed, BASIC PRINTs a question mark.

Here's an example:

```
10 REM ** EXAMPLE OF USING AN "INPUT" STATEMENT
20 REM ** TO OBTAIN MANUALLY ENTERED DATA
30 REM
40 REM CALL FOR KEYBOARD INPUT AND PLACE IN 'X' VARIABLE
50 INPUT X
60 REM PRINT THE DATA THAT WAS TYPED
70 PRINT "THE DATA TYPED ON THE KEYBOARD WAS"; X
80 STOP
```

Data typed on the keyboard is underlined in the following PRINT-out of the above program.

```
?1776
THE DATA TYPED ON THE KEYBOARD WAS 1776
```

To obtain manually typed data for use in a BASIC program, the \_\_\_\_\_ Statement is used.

INPUT



12. More than one data item can be called for by a single INPUT Statement. The method is to follow the Statement with a **list** of variables which are to receive the data.

For example:

```
10 REM ** PROGRAM TO INPUT TWO DATA ITEMS FROM KEYBOARD
20 REM
30 REM THE INPUT STATEMENT IS FOLLOWED BY A LIST OF VARIABLES
40 INPUT X,Y
50 REM THE DATA CAN BE PRINTED OR USED IN COMPUTATIONS
60 PRINT X;"TIMES";Y"EQUALS";X*Y
70 STOP
```

When the question mark prompt advises that data are to be typed on the keyboard, a value must be entered for **each** variable in the list. Each data value typed must be separated by a comma. An example of executing the above program is:

?4,25

4 TIMES 25 EQUALS 100

More than one manually-entered data item can be called for if the INPUT Statement is followed by a \_\_\_\_\_ of variables.

list

13. Often, it is desirable to tell the keyboard operator what data should be typed in response to an INPUT Statement. Of course, you know enough about BASIC by now to be able to create a "prompt message" program segment like

```
10 PRINT "WHAT YEAR IS THIS"
20 INPUT Y
30 PRINT "YOU SAY THAT THIS IS THE YEAR";Y
40 STOP
```

This requirement comes up so often that BASIC includes provisions to make it easier to PRINT an INPUT prompt message. Simply follow the INPUT Statement with your own prompt (in quotes); then a semicolon followed by your list of variables. Here's how it would look:

```
10 INPUT "WHAT YEAR IS THIS?";Y
20 PRINT "YOU SAY THAT THIS IS THE YEAR";Y
30 STOP
```

Using the INPUT prompt method in the second example, you would see:

```
WHAT YEAR IS THIS? 1977
YOU SAY THAT THIS IS THE YEAR 1977
```

An INPUT Statement can include a \_\_\_\_\_ to tell the keyboard operator what data is desired.

prompt message

14. Let's rewrite the Score Averaging program so that the scores can be entered from the keyboard.

```
10 REM **SCORE AVERAGING PROGRAM WITH KEYBOARD ENTRY
20 REM
30 REM GET THE FIRST SCORE WITH A SPECIAL PROMPT MESSAGE
40 INPUT "WHAT'S THE FIRST SCORE?";S
50 REM THEN JUMP AROUND INPUT FOR REMAINING SCORES
60 GOTO 100
70 REM REST OF SCORES ARE INPUT BY NEXT LINE
80 INPUT "NEXT SCORE? ";S
90 REM TEST FOR ENTRY OF 999 TO SIGNAL 'DONE'
100 IF S=999 THEN GOTO 170
110 REM GET TOTAL SCORE AND KEEP COUNT
120 LET T=T+S
130 LET N=N+1
140 REM LOOP BACK FOR NEXT SCORE
150 GOTO 80
160 REM PRINT THE ANSWER
170 PRINT "THE AVERAGE OF";N;"SCORES IS";T/N
180 STOP
```

Here's the program being executed:

```
WHAT'S THE FIRST SCORE? 78
NEXT SCORE? 82
NEXT SCORE? 94
NEXT SCORE? 87
NEXT SCORE? 84
NEXT SCORE? 86
NEXT SCORE? 67
NEXT SCORE? 92
NEXT SCORE? 88
NEXT SCORE? 999
THE AVERAGE OF 9 SCORES IS 84.2222
```

The computer ignores everything following a \_\_\_\_\_  
Statement on a program line.

REM or REMark



Finally, we have the "What happens if . . . ." department.

What happens if you don't type as many data items as called for in the INPUT Statement? Answer: BASIC will keep PRINTing a question mark prompt until you have typed in all the data required.

What happens if you type in too many data items? Answer: The extra items will be ignored.

Turn to your Workbook and perform the exercises labeled "Exercise Group 4". Then return to this page.

## REVIEW TEST

1. The argument of a PRINT Statement can consist of multiple items to be PRINTed if the items are arranged in a \_\_\_\_\_.
2. The items in a list following a PRINT keyword must be separated by \_\_\_\_\_ or \_\_\_\_\_.
3. When items in a list to be PRINTed are separated by semicolons, they will be PRINTed on the \_\_\_\_\_ line and without any \_\_\_\_\_ spaces.
4. When items in a list to be PRINTed are separated by commas, they will be PRINTed on the \_\_\_\_\_ line with spaces added to align them into \_\_\_\_\_.
5. BASIC considers a line of printing to be divided into columns, each \_\_\_\_\_ characters wide.
6. The arguments of several PRINT Statements appearing on different program lines can be joined into a single \_\_\_\_\_ if the arguments end with a semicolon or a comma.
7. String literals, variable names, and expressions (can) (cannot) appear in the same list of items to be PRINTed.
8. A Statement pair that can be used to instruct BASIC to assign sequential data values from a list is \_\_\_\_\_ and \_\_\_\_\_.
9. It is often desirable to add an identifiable number as the last value in a DATA list as a \_\_\_\_\_.
10. A "stopper" is added as the last item in a DATA list so that a program test can be made to determine \_\_\_\_\_.
11. The READ Statement has as its argument one or more \_\_\_\_\_.
12. If several DATA Statements appear in a program, BASIC considers them to comprise a \_\_\_\_\_ list.

13. BASIC remembers the position of the last data item READ from the DATA list. When a READ Statement is next encountered, the data will be READ from the \_\_\_\_\_ position in the list.
14. Items in a DATA list are separated by \_\_\_\_\_.
15. If the argument of a READ Statement has several variables, they are separated by \_\_\_\_\_.
16. Attempting to READ more data items than are provided in DATA Statements will result in an \_\_\_\_\_.
17. Comments, headings, and explanatory text can be added to programs to make them easier to understand by using the \_\_\_\_\_ Statement.
18. The computer \_\_\_\_\_ everything which follows a REMark Statement on a program line.
19. Data can be provided to a program from the Console keyboard by using the \_\_\_\_\_ Statement.
20. When the INPUT Statement is encountered in executing a program, BASIC PRINTs a question mark as a \_\_\_\_\_.
21. When several data items are required by an INPUT Statement, the keyboard operator must separate each individual entry with a \_\_\_\_\_.
22. If less data items are typed than the INPUT Statement requires, BASIC will PRINT a \_\_\_\_\_ prompt until it receives enough data items.
23. If more data items are typed than the INPUT Statement requires, BASIC will \_\_\_\_\_ the extra data items.
24. A \_\_\_\_\_ can be substituted for the question mark prompt PRINTed by BASIC when it encounters an INPUT Statement.
25. To cause BASIC to PRINT a prompt message when it encounters an INPUT Statement, the desired message must be enclosed by \_\_\_\_\_ and placed as the first item in the INPUT list.



## REVIEW ANSWERS

| <u>QUESTION</u> | <u>ANSWER</u>                            | <u>FRAME #</u><br><u>FOR REVIEW</u> |
|-----------------|------------------------------------------|-------------------------------------|
| 1.              | List. ....                               | 1                                   |
| 2.              | Commas, semicolons. ....                 | 2 & 5                               |
| 3.              | Same, added. ....                        | 2                                   |
| 4.              | Same, columns. ....                      | 5                                   |
| 5.              | 14. ....                                 | 5                                   |
| 6.              | List. ....                               | 3 & 5                               |
| 7.              | Can. ....                                | 4                                   |
| 8.              | READ, DATA. ....                         | 6                                   |
| 9.              | "Stopper". ....                          | 6                                   |
| 10.             | When all DATA items have been used. .... | 6                                   |
| 11.             | Variables. ....                          | 7                                   |
| 12.             | Single. ....                             | 7                                   |

|     |                       |            |
|-----|-----------------------|------------|
| 13. | Next. ....            | 6          |
| 14. | Commas. ....          | 6          |
| 15. | Commas. ....          | 7          |
| 16. | Error message. ....   | 8          |
| 17. | REMark. ....          | 10         |
| 18. | Ignores. ....         | 10         |
| 19. | INPUT. ....           | 11         |
| 20. | Prompt. ....          | 11         |
| 21. | Comma. ....           | 12         |
| 22. | Question mark. ....   | Conclusion |
| 23. | Ignore. ....          | Conclusion |
| 24. | Prompt message. ....  | 13         |
| 25. | Quotation marks. .... | 13         |

**HEATHKIT  
CONTINUING  
EDUCATION**

# **Individual Learning Program**

## **BASIC PROGRAMMING**

### **Segment 4**

### **BUILD A DOGHOUSE**

**EC-1100**

**HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022**

**Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America**



## Segment 4

# BUILD A DOGHOUSE!

### INTRODUCTION

In the Introduction to this course we referred to the Statements that are used to construct a program as “tools”. As in any endeavor, there is a set of “standard” tools that are essential to the job and a set of “deluxe” tools that can be added to make the work easier. The Statements we have discussed so far in the Course comprise the “standard” tool set of BASIC Language Programming. It may surprise you to learn that virtually any computer program which you might conceive can be implemented using only the Statements that have been described so far.

Of course, there are both “brute force” and sophisticated methods of accomplishing almost any task. While a Cathedral could be built with only “standard” carpenter’s tools, it would be much easier for the carpenter if “deluxe” tools were available. But it could be done! The same is true of the “standard” set of programming tools we have described so far. Monumental programs can be written using these Statements alone.

Let’s apply our simple tools to a simple task and build the “doghouse” that our Introduction named as the starting point. Later, as we acquire some of the “deluxe” tools, we will come back to the doghouse and add a room or two and then a second story.

## UNIT OBJECTIVES

When you have completed Segment 4, you will have participated in the creation of a BASIC program to calculate the value of a stack of U.S. currency, and will be able to:

1. Name the Statement that allows a list of DATA to be READ more than once in a program.
2. Name the Statement that can be used to specify the position on a line of PRINTing where the next character will be PRINTed.
3. Describe the way an argument for a TAB Statement must be typed.
4. Describe the computer's response to several TAB Statements.

## LET'S WRITE A PROGRAM

To write any computer program, these items are required:

1. A **precise definition** of what the program is to accomplish.
2. A **detailed**, step-by-step **plan** of how the program goals will be achieved.
3. **Knowledge** of how the programming Statements can be used to implement the step-by-step plan.

### REQUIREMENT 1

*Precise definition of what the program is to accomplish.*

You have probably heard a comment such as, "A pile of one dollar bills equalling the National Debt would reach to the moon"; or, "A stack of a million one dollar bills would be as high as the Empire State Building". Are these statements true? What other oddities can be discovered about U.S. Currency stacked into a single pile? How much money would it **really** take to make a stack that would reach the moon?

We could answer these questions and more by writing a computer program that would make the required computations. You will find that there are some interesting facts to learn about piles of money when you have a computer execute

### THE STACK OF MONEY PROGRAM

The Stack of Money Program will begin in a simple form. Later, we will add features that illustrate the use of some of the "deluxe" programming Statements being described. In this way, we will be using a familiar base on which to "build" increased capabilities. Here's **exactly** what the program is to accomplish:

Given the denomination of U.S. Currency to be used in the computation and either the height of a stack of the specified Currency or the total value of the stack, the program will compute the unknown factor and PRINT the three factors: denomination, height, and total value.



**REQUIREMENT 2.**

*Step-by-step plan of how the program goals will be achieved.*

The precise definition of what the program is to accomplish is the first and most important part of our creative cycle; but it is often supplied by someone else. The programmer who assembles the available instructions into a program that does the stated job is the one who invokes the magic of the computer.

The exact method of developing the step-by-step plan is different for each person. Some people rely heavily on a technique known as **Flow-Charting**. Others have an intuitive “feel” for program development and write “off the top of their head”. Whatever method you use, it must be honed to a skill instead of a struggle.

Let’s employ a simple version of the Flow-Charting technique for our plan of the Stack of Money program. It is easy to understand and explain, and will serve to focus your thought process on the job to be done.

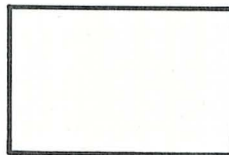
A Flow-Chart is simply a set of distinctively-shaped symbols that indicate the steps of program execution. Flow-Charting could take an entire course by itself, but we are going to keep it very simple.

Here are the symbols we will use:

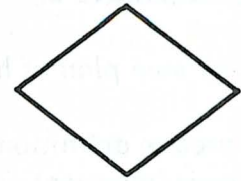
Beginning Point and Ending Point



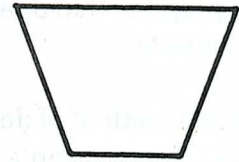
Do Something



Make a Decision



PRINT Something



INPUT Something (from Keyboard)



All we have to do is take the description of exactly what the Stack of Money program is to accomplish and provide the appropriate Flow-Chart symbol for each step.

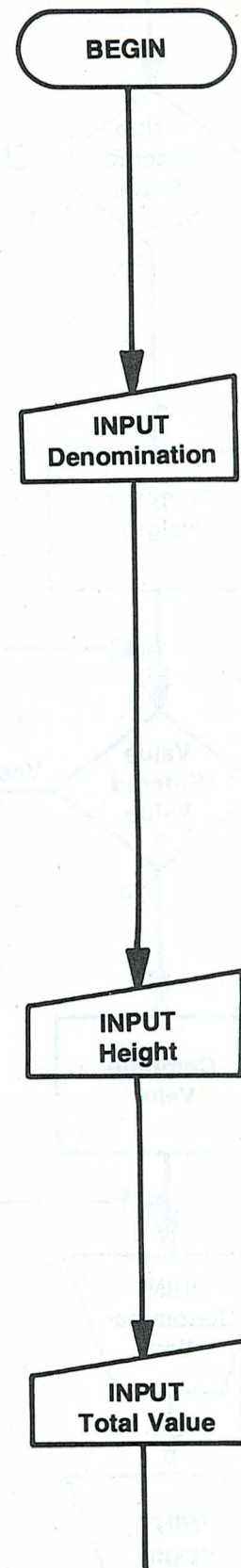


First, each program has to have a Beginning Point

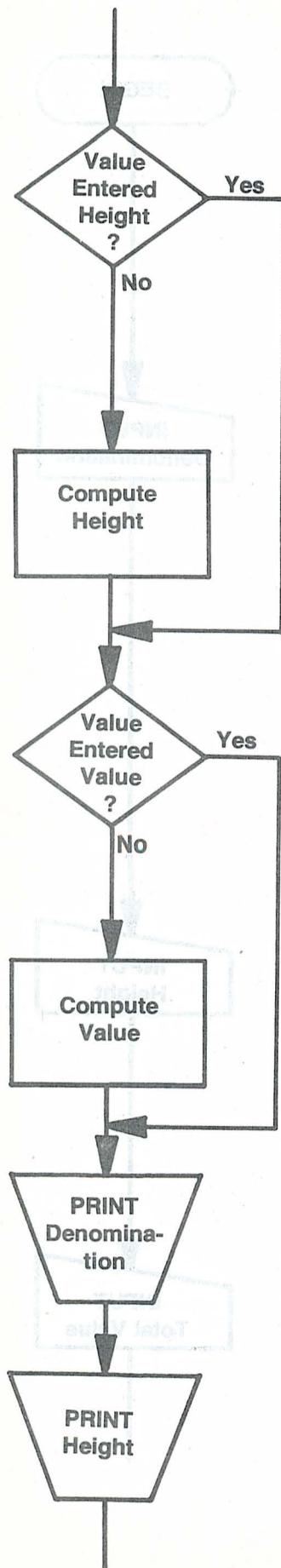
Next, the description says we will be given the denomination of U.S. Currency. If this is to be able to change each time the program is RUN, the information should come from the Keyboard.

Next, we will be given the height of the stack. Keyboard input again.

Next, we will be given the total value of the stack. Keyboard input again.







Now the program has to decide if the height of the stack is to be computed.

If so, do the computation.

Decide if total value is to be computed.

If so, do the computation.

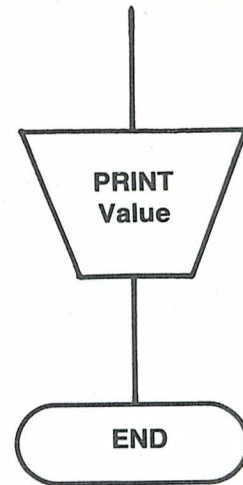
Now PRINT the denomination.

PRINT the height of the stack.

PRINT the total value.

All programs must have an Ending Point.

That's the job we have to do. It doesn't look too hard! Let's proceed to the last item required to write a program.



**REQUIREMENT 3.**

*Implement the step-by-step plan with programming Statements.*

This is the "standard" set of tools that we have discussed:

| Standard Programming Statements |                                                                                         |
|---------------------------------|-----------------------------------------------------------------------------------------|
| LET                             | Assigns names and values to variables.                                                  |
| PRINT <string literal>          | Prints on paper or displays on screen the exact characters enclosed by quotation marks. |
| PRINT <variable name>           | Prints or displays numerical content of variable.                                       |
| PRINT <expression>              | Computes mathematical expression; then prints or displays the result.                   |
| PRINT<item>;<item>              | List of items printed or displayed on same line without added spaces.                   |
| PRINT <item>,< item>            | List of items printed or displayed on same line aligned into columns.                   |
| REM                             | Remarks can be included in program to document or explain. Ignored by computer.         |
| GOTO                            | Alters order of program execution.                                                      |
| IF. . . THEN                    | Makes decisions based on relational tests.                                              |
| READ. . . DATA                  | Supplies data items from program-resident list.                                         |
| INPUT                           | Supplies data items from keyboard.                                                      |
| STOP                            | Stops program execution.                                                                |
| Deluxe Programming Statement    |                                                                                         |
| ON. . . GOTO                    | Program branches made to computed line number.                                          |



Now, let's apply these tools to accomplish the steps shown in the program Flow-Chart.

The first step in the Chart is "Begin". Since nothing is really required to "begin", we will take this opportunity to use a REMark Statement giving the title of the program:

```
10 REM ** THE STACK OF MONEY PROGRAM **
20 REM
```

(The second REMark line merely separates the title from the rest of the program and makes it easier to read.)

The second step in the Chart calls for INPUTting the denomination of the currency to be used in the computation. This data will come from the keyboard. We need to select a **variable** name that will receive the data. How about "D", for denomination?

```
30 INPUT D
```

The next step calls for INPUTting the height of the stack. Again, a variable name must be selected to receive this data. Variable "H", for height, seems to be a good choice.

```
40 INPUT H
```

The last data required from the keyboard is the total value. We will INPUT this information to variable "V".

```
50 INPUT V
```

The next step in the Flow-Chart calls for a decision. If the height of the stack is the unknown quantity that is to be computed, we will establish the rule that "0" is to be entered on the keyboard when INPUT of the height is requested. Thus, if variable H contains zero, we will use denomination and total value to compute the height.

A Relational Test can be used to determine if zero was entered for variable H, and if so, to make the computation for height. Our tool kit contains IF. . . THEN, which is just what we need for this job.

The computation requires us to come up with a mathematical expression. You don't have to be a mathematics genius to write programs — but mathematical computations come up often and the step-by-step logical approach serves as well here as when writing programs.

To construct an expression for the height of the stack, we will take the total value (variable V) and divide it by the denomination of the currency (variable D). This result gives us the number of bills in the stack. The formula is simply  $V/D$  (total value divided by denomination).

Knowing how many bills are in the stack, we can find out how many inches high it is by multiplying the thickness of one bill by the number of bills in the stack. Our trusty micrometer measures the average thickness of several dollar bills as 0.005 inch, so  $(V/D) * .005$  is the stack height in inches. Now, with the height of the stack computed in inches, we can convert to feet. Feet, of course, are inches divided by 12; thus,  $((V/D) * .005) / 12$  gives the desired answer in feet. Assigning the result of this computation to variable H simply requires that we put "LET H=" in front of the expression.

```
60 IF H=0 THEN LET H=((V/D)*.005)/12
```

It looks complicated when you are finished but, as you can see, it is really a series of simple steps.

The next Flow-Chart step calls for a similar computation if the total value variable (V) contains zero. Another expression must be constructed to compute the total value of the stack when the denomination and height are known. Simply find how many bills are in the stack and multiply by the denomination.

Convert the stack height, in feet, to inches.  $H * 12$  will do it. Then divide that result by the thickness of one bill to find how many bills are in the stack.  $(H * 12) / .005$  is the way. Then multiply the denomination of the bill by the number of bills in the stack with the expression  $D * ((H * 12) / .005)$ . Add the LET "V=" to assign the computed result to the V variable and you are done.

```
70 IF V=0 THEN LET V=D*((H*12)/.005)
```

The next three steps in the Flow-Chart call for PRINTing the results of the computations.

```
80 PRINT D
90 PRINT H
100 PRINT V
```

The last Flow-Chart step is the ending point. For BASIC, this is

```
110 STOP
```

Here is the initial version of our program:

```
10 REM ** THE STACK OF MONEY PROGRAM **
20 REM
30 INPUT D
40 INPUT H
50 INPUT V
60 IF H=0 THEN LET H=((V/D)*.005)/12
70 IF V=0 THEN LET V=D*((H*12)/.005)
80 PRINT D
90 PRINT H
100 PRINT V
110 STOP
```

And here is how it will look when executed:

```
?1
?1
?0
1
1
2400
```

Which tells us that a stack of one dollar bills, one foot high, is worth \$2400.



## MAKE IT BETTER

If you have a feel for “showmanship”, you can probably see some deficiencies in The Stack of Money program; particularly in the appearance of the data entry and result PRINTout portion. The way data is requested and the presentation of computation results is called “formatting”, and our example program leaves a lot to be desired in these important areas. There is no identification of the data the operator is requested to INPUT, and the PRINTout does not specify what data is being presented. If these omissions bother you, congratulations! The time to add fancy features, however, is after you have written your program in the simplest possible form that will accomplish the objectives.

Until you have developed your own “style” of programming, we urge you to create the program in its most basic form; then work out any errors that may exist before adding the “gingerbread”. It is very easy to work with BASIC in this manner. As the Stack of Money program demonstrates, the performance of the program can be measured against the original objectives in its simplest form. When you know it works, you can add fancy features and test them as you go. It’s easier to focus your attention on accomplishing the objectives with the simple approach, and it’s easier to find mistakes, too.

## FORMATTING

Now that the Stack of Money program does what we set out to do, let’s add some **formatting** to make the INPUT and PRINTout portions look nicer. The first and most obvious problem is that the keyboard operator does not know what data the program is requesting at each of the three INPUT operations. Here is where the **INPUT prompt message** feature can be used.

You can change the following program lines by simply typing the old line number and the new Statements. When you type in a program line, it replaces an old line, if one exists, with the same number.

```
30 INPUT "DENOMINATION OF BILL? ";D
40 INPUT "HEIGHT OF STACK IN FEET? ";H
50 INPUT "TOTAL VALUE OF STACK? ";V
```

Here's the way the modified program works:

```
DENOMINATION OF BILL? 1
HEIGHT OF STACK IN FEET? 1
TOTAL VALUE OF STACK? 0
1
1
2400
```

Quite an improvement!

Let's attack the PRINT out of the computed results. As you can see, there is no identification of what the PRINTed data represents; so we will add some:

```
80 PRINT "A STACK OF";D;"DOLLAR BILLS"
90 PRINT H;"FEET HIGH"
100 PRINT "IS WORTH";V;"DOLLARS"
```

And the result:

```
DENOMINATION OF BILL? 1
HEIGHT OF STACK IN FEET? 1
TOTAL VALUE OF STACK? 0
A STACK OF 1 DOLLAR BILLS
1 FEET HIGH
IS WORTH 2400 DOLLARS
```

It's getting better all the time!

Since the INPUT and computed results run together, it would be better if they were separated by a blank line.

Here is where you make good use of the fact that you have left some numbers available between lines of your program. We have room to add a blank PRINT Statement. The Statement PRINT without an argument causes a blank line to be PRINTed.

```
75 PRINT
```

We can also use another Statement from our standard tool kit to correct BASIC's grammar! It should know better than to say, "1 FEET". The IF... THEN Statement pair can be used to test the content of the H variable which contains the height of the stack in feet. If the height is one or less than one, the correct word is "FOOT", not "FEET".

```
82 IF H>1 THEN GOTO 90
84 PRINT H;"FOOT HIGH"
86 GOTO 100
```

With these program lines, line 90 will be executed if the stack is more than 1 foot high, and line 84 will be executed if it is 1 foot or less.

```
DENOMINATION OF BILL? 1
HEIGHT OF STACK IN FEET? 1
TOTAL VALUE OF STACK? 0
```

```
A STACK OF 1 DOLLAR BILLS
 1 FOOT HIGH
IS WORTH 2400 DOLLARS
```

Looks nicer than when we started, doesn't it?

## GOOF-PROOFING

Ever see someone touch their finger to a wall right under a sign that says, "WET PAINT"? Maybe you have a friend that would push a button marked, "DON'T PUSH THIS BUTTON", just to "see what would happen". We have such a friend; his name is "Watson".

According to a man named Murphy, "That which can possibly happen, will." Computer programmers must be constantly mindful of Mr. Murphy and his law, and carefully evaluate their programs for the proper placement of "anti-Watson's".

The Stack of Money program could use several anti-Watson's, some of which are obvious and some which were discovered only after inviting Mr. Watson to try it out. The very first thing he did was to enter a three dollar bill! Of course, BASIC didn't know the difference and calculated a 1 foot high stack of 3 dollar bills as being worth 7200 dollars. While it is relatively unimportant if you are able to fool the Stack of Money program, it might be critical in other programs to assure **valid** computations.



The READ . . . DATA Statement pair is perfect for checking that the denomination entered is a "legal" one. Our Bicentennial Edition of The World Almanac & Book of Facts tells us that the only denominations of U.S. currency in circulation for use by the general public are: \$1,2,5,10,20,50,100. It would be easy to check the keyboard denomination entry against this list to see if it was "legal":

```
33 READ A
34 IF D=A THEN GOTO 40
36 IF A<>999 THEN GOTO 33
37 PRINT "THERE IS NO";D;"DOLLAR BILL"
38 GOTO 30
108 DATA 1,2,5,10,20,50,100,999
```

If you add the above lines and try the Stack of Money program, you will get an !ERROR - NO DATA AT LINE 33 after trying to enter an "illegal" denomination. The reason this occurs is that BASIC keeps track of a "pointer" to the next DATA value to be used. After checking through the entire DATA list for an "illegal" denomination, the pointer is at the end of the list and no more DATA can be READ.

BASIC includes a Statement that resets the pointer to the beginning of the DATA list: RESTORE. By adding this Statement just before our test for "illegal" denominations, we can READ the DATA list as many times as necessary. Note that RESTORE can be used even when the pointer is already at the beginning of the DATA list.

```
32 RESTORE
```

Now, here's how our program works:

```
DENOMINATION OF BILL? 3
THERE IS NO 3 DOLLAR BILL
DENOMINATION OF BILL? 8
THERE IS NO 8 DOLLAR BILL
DENOMINATION OF BILL? 1
HEIGHT OF STACK IN FEET? 2
TOTAL VALUE OF STACK? 0
```

```
A STACK OF 1 DOLLAR BILLS
2 FEET HIGH
IS WORTH 4800 DOLLARS
```

Let's add a little "showmanship".

When an "illegal" denomination is entered, the error message and repeat of the denomination request are PRINTed on successive lines. The messages all run together and are hard to read. By adding a blank PRINT line, there will be a separation that will improve the appearance. First, add the blank:

```
29 PRINT
```

Then change the destination of the error GOTO:

```
38 GOTO 29
```

```
DENOMINATION OF BILL? 3
THERE IS NO 3 DOLLAR BILL
```

```
DENOMINATION OF BILL? 1
HEIGHT OF STACK IN FEET? 1
TOTAL VALUE OF STACK? 0
```

```
A STACK OF 1 DOLLAR BILLS
1 FOOT HIGH
IS WORTH 2400 DOLLARS
```

Here's a new Statement that you can use to improve formatting: TAB. This Statement works just like the Tab Stops on a typewriter; that is, it "moves" the position where the next character will be PRINTed to a specific place on the line. The argument portion of the TAB Statement specifies how many places from the left side of the line the next character is to PRINT. Keep in mind that TAB's are not cumulative. That is, you do not specify how many places to move from the last place; the number of places to move are always counted from the beginning of the line. (TAB does **not** cause a new line of PRINTing to begin.)

The way the argument of a TAB Statement is typed is different from the other Statements we have discussed. This is because TAB is a function, and the BASIC Interpreter Program "reads" this type of Statement in a different way. There are quite a number of function Statements which we will be describing in a later Segment, but for now, just remember that arguments for functions must be enclosed in parentheses, such as TAB(20).

Just like other BASIC Statements, arguments for functions can consist of any mix of real numbers, variable names and computation results. For example, TAB(3\*4+1) is OK.

If you try to TAB to a lower-numbered position on the line than where the next character would normally be PRINTed, or if you try to TAB beyond the end of the line, the TAB Statement will be ignored.

Let's apply the TAB Statement to the Stack of Money program to dress up the PRINTed results a bit. How about these changes:

```
37 PRINT TAB(5) "THERE IS NO";D;"DOLLAR BILL"
84 PRINT TAB(5) H;"FOOT HIGH"
90 PRINT TAB(5) H;"FEET HIGH"
100 PRINT TAB(10) "IS WORTH";V;"DOLLARS"
```

And a PRINTout:

```
DENOMINATION OF BILL? 3
 THERE IS NO 3 DOLLAR BILL
```

```
DENOMINATION OF BILL? 1
HEIGHT OF STACK IN FEET? 1
TOTAL VALUE OF STACK? 0
```

```
A STACK OF 1 DOLLAR BILLS
 1 FOOT HIGH
 IS WORTH 2400 DOLLARS
```

Finally, another anti-Watson.

What would happen if you entered zero for both the height of the stack and the total value? Well, BASIC would slavishly compute both values and come up with zero for each. Let's include some Statements to check for this condition.

First, if a value other than zero is INPUT for height, there is no need to ask for an INPUT of total value since that is obviously what is to be computed. Here's a program line to make that test:

```
42 IF H>0 THEN GOTO 60
```



Next, **only** if the height is INPUT as zero will we execute line 50. If we get a zero for V, then we know Watson is at work. These program lines will check it out:

```
54 IF V>0 THEN GOTO 60
56 PRINT TAB(5) "THAT'S NOT ENOUGH INFORMATION"
58 GOTO 29
```

Now we have covered all possibilities; here's the program listing with all the additions discussed above plus some additional REMarks to explain how it works:

```
10 REM ** THE STACK OF MONEY PROGRAM **
20 REM
24 REM ASK FOR DENOMINATION OF BILL
29 PRINT
30 INPUT "DENOMINATION OF BILL? ";D
31 REM CHECK FOR LEGAL DENOMINATION
32 RESTORE
33 READ A
34 IF D=A THEN GOTO 40
36 IF A<>999 THEN GOTO 33
37 PRINT TAB(5) "THERE IS NO";D;"DOLLAR BILL"
38 GOTO 29
39 REM ASK FOR HEIGHT OF STACK
40 INPUT "HEIGHT OF STACK IN FEET? ";H
41 REM CHECK IF VALUE IS TO BE COMPUTED
42 IF H>0 THEN GOTO 60
49 REM IF HEIGHT IS TO BE COMPUTED GET VALUE
50 INPUT "TOTAL VALUE OF STACK? ";V
52 REM CHECK FOR DOUBLE ZERO ENTRY
54 IF V>0 THEN GOTO 60
55 REM PRINT ERROR MESSAGE
56 PRINT TAB(5) "THAT'S NOT ENOUGH INFORMATION"
```

```
58 GOTO 29
59 REM MAKE COMPUTATIONS
60 IF H=0 THEN LET H=((V/D)*.005)/12
70 IF V=0 THEN LET V=D*((H*12)/.005)
74 REM PRINT RESULTS
75 PRINT
80 PRINT "A STACK OF";D;"DOLLAR BILLS"
81 REM CHECK WHETHER TO PRINT FOOT OR FEET
82 IF H>1 THEN GOTO 90
84 PRINT TAB(5) H;"FOOT HIGH"
86 GOTO 100
90 PRINT TAB(5) H;"FEET HIGH"
100 PRINT TAB(10) "IS WORTH";V;"DOLLARS"
105 REM HERE IS THE LIST OF 'LEGAL' BILLS
108 DATA 1,2,5,10,20,50,100,999
109 REM GOODBYE! HOPE YOU ENJOYED THIS PROGRAM
110 STOP
```

Here's a RUN of the finished version:

```
DENOMINATION OF BILL? 3
 THERE IS NO 3 DOLLAR BILL

DENOMINATION OF BILL? 1
HEIGHT OF STACK? 0
TOTAL VALUE OF STACK? 0
 THAT'S NOT ENOUGH INFORMATION

DENOMINATION OF BILL? 1
HEIGHT OF STACK? 1

A STACK OF 1 DOLLAR BILLS
 1 FOOT HIGH
 IS WORTH 2400 DOLLARS
```





## REVIEW TEST

1. A list of DATA can be READ more than once in a program if the Statement \_\_\_\_\_ is used to reset the "pointer" to the beginning of the list.
2. The position on a line where the next character will be PRINTed can be specified by the \_\_\_\_\_ Statement.
3. The argument for the Statement in question 2 must be \_\_\_\_\_.
4. The Statement TAB(5) will cause the next character to be PRINTed 5 places to the right of the last character that was PRINTed. (true) (false)
5. The Statement TAB(5) will cause the next character to be PRINTed 5 places to the right of the beginning of the line. (true) (false)
6. If the last character was PRINTed at position 20 on the line, the Statement TAB(10) will cause what to happen? \_\_\_\_\_.
7. If the longest line that can be PRINTed is 80 characters, the Statement TAB(100) will cause what to happen? \_\_\_\_\_.

## REVIEW TEST

A list of TAA can be READ more than once as a program if the statement is used to read the "pointer" to the beginning of the list.

The position on a line where the next character will be PRINTed can be specified by the statement:

The argument for the statement in question 2 must be:

The statement TAB(2) will cause the next character to be PRINTed 2 places to the right of the last character that was PRINTed (true) (false)

The statement TAB(2) will cause the next character to be PRINTed 2 places to the right of the beginning of the line (true) (false)

If the last character was PRINTed at position 20 on the line, the statement TAB(10) will cause what to happen?

If the longest line that can be PRINTed is 60 characters, the statement TAB(50) will cause what to happen?

## REVIEW ANSWERS

1. RESTORE.
2. TAB.
3. Enclosed in parentheses.
4. False.
5. True.
6. Nothing. The TAB Statement will be ignored.
7. Nothing. The TAB Statement will be ignored.





**HEATHKIT  
CONTINUING  
EDUCATION**

# **Individual Learning Program**

## **BASIC PROGRAMMING**

*Segment 5*

**HOW FUNCTIONS FUNCTION!**

**EC-1100**

**HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022**

**Copyright © 1977**  
Heath Company  
*All Rights Reserved*  
Printed in the United States of America

## Segment 5

# HOW FUNCTIONS FUNCTION!

### INTRODUCTION

In Segment 4 we said that the Statement "TAB" was a "Function." In this Segment we will discuss the way Functions work and introduce some additional Function Statements.



## UNIT OBJECTIVES

When you have completed Segment 5, you will be able to:

1. Supply the name for an often-used series of instruction steps which can be called upon as many times as required by the main program.
2. Name the Statement with which all sub-programs must end.
3. Name the Statement which causes the instruction steps of a sub-program to be performed.
4. State what the argument portion of a GOSUB Statement specifies.
5. State where program execution jumps when execution of a sub-program is complete.
6. Identify the category of programs to which Functions belong.
7. State two ways that Functions differ from regular sub-programs.
8. Describe how the argument supplied after a Function name must be formatted.
9. Name the items which may comprise the argument for a Function call.
10. Name the two types of Functions.
11. Match a list of Intrinsic Function names with the proper description of the operation performed.
12. Name the Statement which specifies that a User Function is being defined.
13. State the two parts that comprise a User Function name.
14. State the purpose of dummy arguments in the definition of User Functions.
15. Specify what happens to the real arguments supplied with a call to a User Defined Function.

## PROGRAMMED INSTRUCTION

1. Suppose you sat down to write a program that required a number of computations involving the square roots of numbers. Each time your program required a square root you could perform the necessary computation, but you would find yourself writing the same series of instruction steps each time. This method would be inefficient from the standpoint of typing time and program memory usage.

BASIC provides a way to include an often-used series of instruction steps as a “**sub-program**” which can be called upon as many times as required by the main program.

An often-used series of instruction steps which can be called upon as many times as required by the main program is called a \_\_\_\_\_.

sub-program

2. The Statement pair that makes it possible to call upon a sub-program at a number of points in the main program is

GOSUB . . . . RETURN

The only rule to remember when you are writing a sub-program is that the last Statement of the sub-program must be RETURN.

All sub-programs must end with the Statement \_\_\_\_\_.

RETURN

3. Each time the main program requires the instruction steps of the sub-program to be performed, the keyword GOSUB is written, followed by the first line number of the sub-program. When BASIC reads the Statement GOSUB, it "remembers" the line number on which it appeared. Program execution then jumps to the line number specified as the argument of the GOSUB Statement. When BASIC reads the keyword RETURN at the end of the sub-program, it recalls the line number that it previously "remembered" and jumps back to the next Statement after GOSUB. This is why you can GOSUB at several points in the main program and always RETURN to the proper place.

The argument of a GOSUB Statement specifies the first line number of the desired \_\_\_\_\_.

sub-program



4. Here's an example of a sub-program which might be written to extract the square root of a number:

```

500 REM **PROGRAM TO EXTRACT SQUARE ROOTS **
510 REM
520 REM THE SQUARE ROOT OF THE NUMBER IN
530 REM VARIABLE 'N' IS EXTRACTED.
540 REM
550 REM ESTABLISH INITIAL TEST VALUE AND INITIAL INCREMENT
560 LET S=1
570 LET M=10
580 REM DOES 'S' SQUARED EQUAL OUR NUMBER?
590 IF S*S=N THEN GOTO 740
600 REM HAVE WE GONE TOO FAR?
610 IF S*S>N THEN GOTO 660
620 REM NOT FAR ENOUGH = INCREASE 'S' BY INCREMENT VALUE
630 LET S=S+M
640 GOTO 590
650 REM 'S' SQUARED IS TOO LARGE = SUBTRACT INCREMENT VALUE
660 LET S=S-M
670 REM REDUCE INCREMENT VALUE ONE DECIMAL PLACE
680 LET M=M/10
690 REM IF 5 DECIMAL RESOLUTION REACHED, CALL IT DONE!
700 IF M<.00001 THEN GOTO 740
710 REM IF NOT 5 DECIMAL PLACES, GO TRY AGAIN
720 GOTO 630
730 REM SQUARE ROOT HAS BEEN FOUND = RETURN WITH ANSWER
740 RETURN

```

This is an example of how the Square Root program might be called upon by the main program, using a GOSUB Statement:

```

10 REM **DEMONSTRATION OF GOSUB RETURN USE **
20 REM
30 REM ASK FOR KEYBOARD INPUT OF NUMBER
40 INPUT "SQUARE ROOT OF WHAT NUMBER? ";N
50 REM GO TO SUB-PROGRAM TO CALCULATE SQUARE ROOT
60 GOSUB 500
70 REM PROGRAM RETURNS TO THIS LINE AFTER CALCULATION
80 REM PRINT ANSWER
90 PRINT "THE SQUARE ROOT OF";N;"IS";S
100 STOP

```

----- cont'd. -----



**4. cont'd.**

Here's how our program looks when you RUN it:

```
SQUARE ROOT OF WHAT NUMBER? 25
THE SQUARE ROOT OF 25 IS 5
```

The Statement which causes a sub-program to be executed is \_\_\_\_\_.

**GOSUB**

There are "shortcut" methods of extracting square roots, but our example program is more straightforward and easier to understand. Remember, we are teaching computer programming, not mathematics.

Because of some subtle differences in the way BASIC handles numbers larger than 10,000, the Square Root program may get "stuck" when you try it. Typing CTRL/C will "un-stick" the program. We will discuss number formats in a later Segment.

5. While this Segment is primarily about Functions, we have begun with a discussion of GOSUB . . . . RETURN so that it will be easier to understand how Functions work.

Several mathematical computations would be difficult to implement using the standard BASIC Statements we have learned so far. In particular, you would find it challenging, to say the least, to write programs to compute trigonometric functions. Because of this, and because some computations such as square root are required quite often, BASIC includes several “built-in” sub-programs. When a sub-program can be called upon by name, instead of by line number as demonstrated in Frames 1 through 4, it is called a “Function”.

A sub-program which can be called upon by name is called a \_\_\_\_\_.

Function

6. The two types of Functions which BASIC includes are called “Intrinsic Functions” and “User Functions”. Intrinsic Functions are those that already exist in BASIC, such as “TAB”, and can only be used “as is”. User Functions are those which you define yourself as part of your program.

Functions, then, are a special kind of sub-program. They are similar to the sub-programs we discussed in Frames 1 through 4 because they cause the computer to perform a special process. They are different from the sub-programs we discussed in two respects:

1. They are called upon by specifying the name of the Function instead of using a GOSUB Statement.
2. You must supply an argument after the Function name, such as TAB(10).

A Function is called upon by specifying its \_\_\_\_\_ and supplying an \_\_\_\_\_.

name, argument

7. Just as with the expressions we have previously discussed, the argument for a Function can include any mix of real numbers, computations, and variable names; even other Functions!

The argument for a Function is always enclosed in parentheses, such as TAB(10).

One of BASIC's Intrinsic Functions extracts the square root of its argument. The name of the Function is SQR. To illustrate what a powerful feature Functions are, here's how our GOSUB demonstration program would look when the SQR Function is used instead of our Square Root sub-program.

```
10 REM ** DEMONSTRATION OF INTRINSIC FUNCTION 'SQR' **
20 REM
30 REM ASK FOR KEYBOARD INPUT OF NUMBER
40 INPUT "SQUARE ROOT OF WHAT NUMBER? ";N
50 REM PRINT ANSWER, 'SQR' FUNCTION PROVIDES SQUARE ROOT
60 PRINT "THE SQUARE ROOT OF";N;"IS";SQR(N)
70 STOP
```

An example RUN:

```
SQUARE ROOT OF WHAT NUMBER? 25
THE SQUARE ROOT OF 25 IS 5
```

Arguments for Functions are always enclosed in \_\_\_\_\_.

parentheses



8. When you are calling upon a Function to make a computation, you must also specify what is to be done with the result. For example, the Statement

```
10 SQR(25)
```

would be meaningless, since BASIC would have no way of knowing what you wanted done with the answer.

Examples of proper Function "calls" would be:

```
LET A=SQR(25)
PRINT SQR(25)
ON SQR(A) GOTO 100,200,300,400,500
PRINT SQR ((A*A)+(B*B))
```

Functions are sub-programs which may be executed by specifying the \_\_\_\_\_ of the Function.

name

The following Frames will describe a number of "standard" Functions. Please keep in mind that minor differences exist among BASICs implemented for different computers. Considerations such as available computer memory determine the features of BASIC that can be included. Not all versions of BASIC will have all the Functions described here; some will have more. The User's Guide included with the version of BASIC you are using is the best source of information about which Functions and Statements are available.

In the discussions that follow, we will use a conventional notation to indicate the argument of the Function being described. The simple numeric variable 'X' will be specified as the argument, for example: ABS(X). Keep in mind that 'X' can be any mix of real numbers, variable names, computations, or other Functions.



## 9. FUNCTIONS: SIN(X) COS(X) TAN(X) ATN(X)

These trigonometric Functions provide the tools to compute the angles and sides of triangles.

The circular, or **natural**, system of angular measurement used by BASIC is sometimes called "radian measure" or " $\pi$  measure". The unit of measure is the **radian**.

Computations involving angular degrees can easily be made using the following conversions:

To convert radians to degrees, multiply radians by 57.3.

To convert degrees to radians, multiply degrees by .01745.

While these conversions are sufficiently accurate for most applications, absolute accuracy can be achieved as follows:

To convert radians to degrees, multiply radians by  $180^\circ / \pi$

To convert degrees to radians, multiply degrees by  $\pi / 180^\circ$

To illustrate the use of BASIC's trigonometric functions, here is a program to PRINT a table of natural trigonometric functions for angles of 10 to 20 degrees. Taking the arctangent of the computed tangent validates the computations. Note the conversions between degrees and radians.

```
10 REM ** NATURAL TRIGONOMETRIC FUNCTIONS PROGRAM **
20 REM
30 REM PRINT HEADING FOR TABLE
40 PRINT TAB(6); "TABLE OF NATURAL TRIGONOMETRIC FUNCTIONS"
50 PRINT
60 PRINT "DEGREES", "SINE", "COSINE", "TANGENT", "ARCTANGENT"
70 REM START WITH 10 DEGREES
80 LET N=10
90 REM CONVERT TO RADIANS
100 LET X=.01745*N
110 REM PRINT FUNCTIONS OF 'X'
120 PRINT N, SIN(X), COS(X), TAN(X),
```

--- cont'd. ---

## 9. cont'd.

```

130 REM TAKE ARCTANGENT OF COMPUTED TANGENT AND CONVERT
140 PRINT ATN(TAN(X)) * 57.3
150 REM SET UP FOR NEXT ANGLE
160 LET N=N+1
170 REM TEST TO SEE IF DONE - IF NOT, LOOP FOR NEXT PRINT
180 IF N<21 THEN GOTO 100
190 REM ALL DONE - GOODBYE
200 STOP

```

Here is a PRINTout of our Table:

TABLE OF NATURAL TRIGONOMETRIC FUNCTIONS

| DEGREES | SINE    | COSINE  | TANGENT | ARCTANGENT |
|---------|---------|---------|---------|------------|
| 10      | .173616 | .984814 | .176293 | 9.99884    |
| 11      | .190773 | .981634 | .194343 | 10.9987    |
| 12      | .207873 | .978156 | .212515 | 11.9986    |
| 13      | .224909 | .97438  | .230823 | 12.9985    |
| 14      | .241877 | .970307 | .249279 | 13.9984    |
| 15      | .258771 | .965939 | .267896 | 14.9983    |
| 16      | .275587 | .961276 | .286688 | 15.9982    |
| 17      | .292318 | .956321 | .305669 | 16.998     |
| 18      | .30896  | .951075 | .324854 | 17.9979    |
| 19      | .325509 | .945539 | .344257 | 18.9978    |
| 20      | .341958 | .939715 | .363895 | 19.9977    |

The unit of measure used by BASIC for trigonometric Functions is the \_\_\_\_\_.

radian

## 10. FUNCTIONS: LOG (X) EXP (X)

The natural system of logarithms, used in many scientific calculations, has as its base:

$$e=2.71828$$

BASIC uses this base for its logarithmic Functions. Conversions to common logarithms can be made as follows:

To convert  $\log_e X$  to  $\log_{10} X$ , multiply  $\log_e X$  by .4343

To convert  $\log_{10} X$  to  $\log_e X$ , multiply  $\log_{10} X$  by 2.3026

The Function LOG(X) returns the logarithm (both characteristic and mantissa) to the **base e** of the argument.

The Function EXP(X) returns the **antilogarithm** of the argument.

The following program demonstrates the use of these Functions. The antilog, which is taken of the computed logarithm, validates the computation and is accurate to three decimal places. (Rounding errors are responsible for the slight discrepancies.) Note the use of a double comma in line 140 to tab over a column.

```
10 REM **DEMONSTRATION PROGRAM FOR LOGARITHMIC FUNCTIONS **
20 REM
30 REM PRINT TABLE HEADING
40 PRINT TAB(6)"TABLE OF LOGARITHMS TO THE BASE E"
50 PRINT
60 PRINT "NUMBER", "LOG E OF NUMBER", "ANTILOG"
70 REM ESTABLISH BEGINNING NUMBER
80 LET X=10
90 REM COMPUTE LOGARITHM
100 LET L=LOG(X)
110 REM COMPUTE ANTILOGARITHM
120 LET A=EXP(L)
130 REM PRINT COMPUTATION RESULTS
140 PRINT X,L,,A
150 REM SET UP FOR NEXT NUMBER
160 LET X=X+1
170 REM TEST TO SEE IF DONE - IF NOT, LOOP FOR NEXT PRINT
```

----- cont'd. -----



**10. cont'd.**

```

180 IF X<21 THEN GOTO 100
190 REM ALL DONE - GOODBYE
200 STOP

```

Here's a PRINTout of our Log Table:

TABLE OF LOGARITHMS TO THE BASE E

| NUMBER | LOG E OF NUMBER | ANTILOG |
|--------|-----------------|---------|
| 10     | 2.30259         | 10      |
| 11     | 2.3979          | 11.0001 |
| 12     | 2.48491         | 12      |
| 13     | 2.56495         | 13      |
| 14     | 2.63905         | 13.9999 |
| 15     | 2.70805         | 15      |
| 16     | 2.77259         | 16      |
| 17     | 2.83321         | 17      |
| 18     | 2.89037         | 17.9999 |
| 19     | 2.94444         | 18.9999 |
| 20     | 2.99573         | 20      |

BASIC uses the (natural) (common) system of logarithms.

natural

## 11. FUNCTION: ABS(X)

Some computations, such as square roots and logarithms, cannot handle negative numbers. The logarithm of  $-400$ , for example, is called an "imaginary logarithm" which cannot be used in computation. However, if all numbers are considered as positive when you are computing logarithms, it is only necessary to affix the proper sign to the result. The function  $\text{ABS}(X)$  converts the argument to a positive value which can be used as required.

For example:

```
10 LET X=15
20 GOSUB 60
30 LET X=-15
40 GOSUB 60
50 STOP
60 PRINT "THE LOGARITHM OF ";X;"IS";LOG(ABS(X))
70 RETURN
```

will PRINT:

```
THE LOGARITHM OF 15 IS 2.70805
THE LOGARITHM OF -15 IS 2.70805
```

If 'X' is equal to  $-1977$ , then  $\text{ABS}(X)$  is \_\_\_\_\_.

1977 or +1977

## 12. FUNCTION: SGN(X)

After computing the logarithm of a negative number using the ABS Function described in Frame 11, you can determine the proper **sign** to affix to the result by testing the original argument for its sign. The Function, SGN(X) returns a number that describes the sign of the argument as follows:

If X is positive, then  $\text{SGN}(X)=+1$

If X is zero, then  $\text{SGN}(X)=0$

If X is negative, then  $\text{SGN}(X)=-1$

Here's how SGN(X) might be used to provide the proper result in the example program of Frame 11:

```
10 LET X=15
20 GOSUB 60
30 LET X=-15
40 GOSUB 60
50 STOP
60 PRINT "THE LOGARITHM OF ";X;"IS";SGN(X)*LOG(ABS(X))
70 RETURN
```

and a PRINTout:

```
THE LOGARITHM OF 15 IS 2.70805
THE LOGARITHM OF -15 IS -2.70805
```

This works because  $\text{SGN}(-15)$  is  $-1$ , and  $-1*2.70805$  is  $-2.70805$ .

If 'X' is equal to  $-1977$ , then  $\text{SGN}(X)$  is \_\_\_\_\_.



## 13. FUNCTION: INT(X)

The **Integer** Function is used when it is necessary to discard the decimal portion of the argument. For example:

INT(3.14159) returns the value 3

Note particularly that the INT function does not "round" the argument — it discards the entire decimal value. Thus:

INT(3.14159) returns the value 3

and

INT(3.99999) returns the value 3

The program:

```
10 PRINT INT(1977/4)
20 STOP
```

will PRINT: \_\_\_\_\_

## 14. FUNCTION: RND(X)

There are many occasions where it is necessary to depart from the precise computations of the computer and come up with an imprecise or “**random**” number. Examples that come to mind are card and dice game simulations. BASIC includes a Function that can be called upon to produce a number “at random”.

Random numbers generated by computers are called “pseudo-random” numbers because the computer can never abandon its role of perfection. Statistically, each number generated by this Function bears a random relationship to any other number, but remember — it is a computer that is choosing the number and it simply cannot do anything in a random fashion.

The random number generator begins its sequence with a specific number, called the “seed”. It shuffles the seed, rearranges its digits, pulls it this way and that and “massages” it until it is satisfied with the result which it delivers as a “random” number. But — give it the same “seed” and it will come up with the same exact sequence of numbers every time. Moral: Don’t play poker for money against a computer!

The argument supplied to the Function call is used to modify the sequence of random numbers generated:

WHEN X ISRND(X) PRODUCES

&lt;0

A random number using X as the new seed

=0

The last random number generated

&gt;0

The next random number

----- cont'd. -----

## 14. cont'd.

The numbers generated by the RND function are always **less** than 1.

For example:

```
10 PRINT RND(1)
20 STOP
```

Might PRINT:

.346069

This allows you to select the magnitude of a random number by multiplication. Suppose you wanted your random numbers to be in the range 0-99. You could write:

```
10 REM FIRST, PUT RANDOM NUMBER IN 'N' VARIABLE
20 LET N=RND(1)
30 REM THEN, MULTIPLY THE NUMBER BY 100
40 LET N=100*N
50 REM THEN, DISCARD ANY DECIMAL PORTION
60 LET N=INT(N)
70 REM PRINT THE RESULT
80 PRINT N
90 STOP
```

Here is a single Statement that would accomplish the same result as the above program:

```
10 PRINT INT(100*RND(1))
20 STOP
```

and a sample PRINTout:

76

The function RND(X) produces a \_\_\_\_\_ -random number.

pseudo



15. FUNCTIONS: MIN(X1,X2,X3, . . . . . )  
MAX(X1,X2,X3, . . . . . )

These Functions are useful in comparing the content of several variables or the results of several computations to determine which is the smallest value (MIN) or the largest (MAX). The values to be compared are arranged in a list which is supplied as the argument to the Function.

An example of their use is:

```
10 PRINT "OF THE DIGITS 0 THROUGH 9:"
20 PRINT "THE SMALLEST IS";MIN(0,1,2,3,4,5,6,7,8,9)
30 PRINT "AND THE LARGEST IS";MAX(0,1,2,3,4,5,6,7,8,9)
40 STOP
```

This program will PRINT:

```
OF THE DIGITS 0 THROUGH 9:
THE SMALLEST IS 0
AND THE LARGEST IS 9
```

The arguments of the MIN and MAX Functions consist of a  
\_\_\_\_\_ of expressions.

list

## 16. FUNCTION: PEEK(X)

The PEEK Function provides a means of determining the value stored at address X of the computer's memory. It is primarily used in conjunction with non-BASIC programs. As an example of its use, assume that memory address 76400 octal (32000 decimal) had the byte 100 octal (64 decimal) stored in it. Keeping in mind that, to BASIC, all values are decimal values:

```
10 PRINT PEEK(32000)
20 STOP
```

Would PRINT:

64

Although not called a Function, BASIC includes a companion Statement to PEEK which provides a means of storing a byte of data in a specific memory location. The Statement is POKE. If it were necessary to store the byte 101 octal (65 decimal) at memory location 76401 octal (32001 decimal), here is how you could do it (decimal, remember):

```
10 POKE 32001,65
20 STOP
```

PEEK and POKE are provided to store and retrieve bytes of data directly in the computer's \_\_\_\_\_.

memory

## 17. FUNCTION: PIN(X)

A typical microcomputer, such as the Heathkit Model H8, can address up to 256 "ports" to which external devices that supply data can be connected. These "Input Ports" receive data from keyboards, tape readers, disk memories, and similar equipment.

BASIC normally receives its data by use of the INPUT Statement. However, it is sometimes desirable to "access" one of the Input Ports directly, without using the INPUT Statement. The PIN Function provides this capability.

Suppose the punched tape in a paper tape reader was positioned so that the code 100 octal (64 decimal) was being read, and further, suppose that the paper tape reader were connected to the computer's Input Port 10 octal (8 decimal). The code on the tape could be read by a BASIC program with the Statement

```
10 PRINT PIN(8)
20 STOP
```

This Statement would PRINT:

64

A companion Statement to PIN, although not a Function, is OUT.

In addition to the 256 Input Ports, there are 256 Output Ports to which devices such as CRT Displays and Printers, which receive data, can be connected. The Statement OUT may be used to "output" a byte of data to a selected port directly, without using BASIC's PRINT Statement.

Suppose a high-speed printer were connected to Output Port 11 octal (9 decimal), and that when it received the data byte 14 octal (12 decimal) it would feed out paper until it reached the top of the next page. This function is commonly called "Form-feed", and may be accomplished with the Statement

```
10 OUT 9,12
20 STOP
```

The Function PIN and the Statement OUT give BASIC direct access to the computer's Input and Output \_\_\_\_\_.

ports



## 18. FUNCTION: POS(X)

This Function returns the number of the position on a line where the next character will be PRINTed. Since POS is implemented as a Function, the argument must be supplied to meet BASIC's syntax requirements. However, the argument is not used by the Function and may be any value; even zero.

The number returned by POS will be 1 when the next character will be PRINTed in the first position on a line. Here's an example use of the POS Statement;

```
10 PRINT "NOW IS THE TIME ";
20 LET X=POS(0)
30 PRINT "FOR ALL GOOD MEN"
40 PRINT
50 PRINT X
60 STOP
```

And a PRINTout:

```
NOW IS THE TIME FOR ALL GOOD MEN
```

```
16
```

The POS Function returns the position number where the \_\_\_\_\_ character will be PRINTed on the line.

next

**19. FUNCTION: DEFINE YOUR OWN!**

In addition to the Intrinsic Functions we have described in this Segment, BASIC allows you to DEFINE your own "User Functions". This feature is particularly useful when the same complex computation may be required at several points in a program. User Functions are not "saved" by BASIC for use in other programs, they are DEFINED and used only within the same program.

Once DEFINED, a User Function can be "called" by its **name** and the value or values with which it is to perform its calculations are supplied as arguments. In this regard, User Functions operate in the same manner as the Intrinsic Functions we have discussed.

Both Intrinsic and User Functions are "called" by \_\_\_\_\_.

name

**20. A User Function is usually DEFINED near the beginning of the program, since it cannot be "called" upon to perform its calculation until it has been DEFINED. Although some BASICs differ in this regard, once a User Function has been DEFINED, it cannot be reDEFINED elsewhere in the program.**

The keyword that tells BASIC that you are going to specify a User Function is

DEF

The keyword DEF (for DEFINE) is then followed by the name you have chosen for the Function. All User Function names must begin with the letters FN (for Function). The last part of the Function name may be any legal variable name, such as A, B1, etc. Here's how it looks so far:

DEF FN H

, for example, if you were DEFINING a function which you have decided to name Function H.

The names for User Functions always begin with the letters \_\_\_\_\_, followed by any legal \_\_\_\_\_ name.

FN, variable

21. Let's use a familiar computation for our explanation of User Functions: The Pythagorean theorem. This often-used computation for determining the length of the sides of a right triangle can be stated:

The square of the hypotenuse (C) is equal to the sum of the squares of the other two sides (A and B), or:

$$C^2 = A^2 + B^2 \quad \text{therefore} \quad C = \sqrt{A^2 + B^2}$$

We can DEFINE a User Function that will perform this computation and return the length of the hypotenuse if we supply, as arguments, the length of the other two sides. We will name the Function H.

When DEFINING the Function, we must provide "**slots**" for the arguments that will be supplied when you are calling the Function, and also tell BASIC where the arguments are to be used in the computation. This is done with "**dummy arguments**", which are written into the Function DEFINITION. When the Function is called, the "**real**" arguments are substituted for the dummy arguments by BASIC.

In DEFINING a User Function, dummy arguments are used to provide \_\_\_\_\_ for the real arguments when the Function is called.

slots



22. Our Hypotenuse-solving Function requires two arguments: the other two sides of the triangle. In DEFINING the Function, we will name the dummy arguments A and B. So far, we have:

DEF FN H(A,B)

Now we must construct an expression for the computation that the Function is to make, using the dummy arguments where the real arguments are to appear in the expression. For our example:

DEF FN H(A,B)=SQR((A\*)+(B\*B))

The dummy arguments must also appear in the expression to indicate where the \_\_\_\_\_ arguments are to be used in the computation.

real

23. Now that we have seen how a User Function is DEFINED, here is how it is called by the main program to perform its computation.

```
10 REM ** EXAMPLE OF USER DEFINED FUNCTION **
20 REM
30 REM FIRST, DEFINE THE FUNCTION
40 DEF FN H(A,B)=SQR((A*A)+(B*B))
50 REM THEN STATE A PROBLEM
60 PRINT "A LADDER RESTS AGAINST A VERTICAL WALL."
70 PRINT "IF THE FOOT OF THE LADDER IS 10 FEET FROM"
80 PRINT "THE WALL AND THE LADDER TOUCHES THE WALL"
90 PRINT "24 FEET ABOVE THE GROUND, THE LADDER IS";
100 REM FINALLY, COMPUTE AND PRINT THE SOLUTION
110 PRINT FN H(10,24);"FEET LONG."
120 STOP
```

Here's a PRINTout:

```
A LADDER RESTS AGAINST A VERTICAL WALL.
IF THE FOOT OF THE LADDER IS 10 FEET FROM
THE WALL AND THE LADDER TOUCHES THE WALL
24 FEET ABOVE THE GROUND, THE LADDER IS 26 FEET LONG.
```

A User Function is called by specifying its \_\_\_\_\_ and supplying the same number of \_\_\_\_\_ as there are dummy arguments in the DEFINITION of the Function.

name, arguments

Turn to your Workbook and perform the exercises labeled "Exercise Group 5." Then return to this page.

Now that we have seen how a User Function is defined, here is how it is called by the main program to perform its computation.

```

10 REM ** EXAMPLE OF USER DEFINED FUNCTION **
20 REM
30 REM ***** DEFINE THE FUNCTION *****
40 DEF FN L(A,B) = SQR(A^2+B^2)
50 REM ***** STATE A PROBLEM *****
60 PRINT "A LADDER RESTS AGAINST A VERTICAL WALL."
70 PRINT "IF THE FOOT OF THE LADDER IS 10 FEET FROM
80 THE WALL AND THE LADDER TOUCHES THE WALL
90 24 FEET ABOVE THE GROUND, THE LADDER IS:"
100 REM ***** COMPUTE AND PRINT THE SOLUTION *****
110 PRINT "LADDER IS 24 FEET LONG."
120 STOP

```

Here's a PRINTOUT:

```

A LADDER RESTS AGAINST A VERTICAL WALL.
IF THE FOOT OF THE LADDER IS 10 FEET FROM
THE WALL AND THE LADDER TOUCHES THE WALL
24 FEET ABOVE THE GROUND, THE LADDER IS 24 FEET LONG.

User Function is called by specifying its name and arguments.
Along the same number of line as there are arguments in the definition of the function.

```

Turn to your Workbook and perform the exercises labeled "Exercise Group 5." Then return to this page.



## REVIEW TEST

1. An often-used series of instruction steps which can be called upon as many times as required by the main program is called a \_\_\_\_\_.
2. All sub-programs must end with the Statement \_\_\_\_\_.
3. The Statement that causes the instruction steps of a sub-program to be performed is \_\_\_\_\_.
4. The argument portion of a GOSUB Statement specifies the \_\_\_\_\_ where execution of the sub-program is to begin.
5. When execution of a sub-program is complete, as signified by the Statement RETURN, program execution jumps back to the main program at the Statement following \_\_\_\_\_.
6. A Function is a type of \_\_\_\_\_.
7. One way that Functions differ from regular sub-programs is that they are called by specifying the \_\_\_\_\_ of the Function instead of using the Statement GOSUB.
8. An \_\_\_\_\_ must be supplied after the Function name.
9. The argument supplied after a Function name must be enclosed in \_\_\_\_\_.
10. The argument for a Function call can include any mix of real numbers, computations, variable names and even other \_\_\_\_\_.
11. There are two types of Functions; those that are "built-in" to BASIC are called \_\_\_\_\_.
12. Functions which you DEFine at the time you write a program are called \_\_\_\_\_.

13. Match the following Function names with their proper description:

FUNCTION NAME    DESCRIPTION

|            |                                                                        |
|------------|------------------------------------------------------------------------|
| TAB(X)     | 1. Computes the logarithm to the base e of X.                          |
| SQR(X)     | 2. Returns the lowest value in the list of arguments.                  |
| SIN(X)     | 3. Returns the data byte at Input Port X.                              |
| COS(X)     | 4. Spaces over X places on the line.                                   |
| TAN(X)     | 5. Returns the line position where the next character will be PRINTed. |
| ATN(X)     | 6. Discards the decimal portion of X.                                  |
| LOG(X)     | 7. Returns the square root of X.                                       |
| EXP(X)     | 8. Returns the highest value in the list of arguments.                 |
| ABS(X)     | 9. Returns X as a positive value.                                      |
| SGN(X)     | 10. Returns the data byte at memory location X.                        |
| INT(X)     | 11. Returns the sine of the angle X.                                   |
| RND(X)     | 12. Returns the antilog of X.                                          |
| MIN(X1,X2) | 13. Returns the cosine of the angle X.                                 |
| MAX(X1,X2) | 14. Returns a pseudo-random number.                                    |
| PEEK(X)    | 15. Returns an indicator of the sign of X.                             |
| PIN(X)     | 16. Returns the tangent of angle X.                                    |
| POS(X)     | 17. Returns the arctangent of X.                                       |

14. User Functions are DEFined by the Statement \_\_\_\_\_.
15. All User DEFined Function names begin with the letters \_\_\_\_\_.
16. The second part of User DEFined Functions can be any legal \_\_\_\_\_ name.
17. Dummy arguments in User DEFined Functions denote slots where the \_\_\_\_\_ arguments will appear when the Function is called.
18. The real arguments supplied with a call to a User DEFined Function will be substituted in the expression where the \_\_\_\_\_ arguments appear.

## REVIEW ANSWERS

| QUESTION | ANSWER                    | FRAME #<br>FOR REVIEW |
|----------|---------------------------|-----------------------|
| 1.       | Sub-program. ....         | 1                     |
| 2.       | RETURN. ....              | 2                     |
| 3.       | GOSUB. ....               | 3                     |
| 4.       | Line number. ....         | 3                     |
| 5.       | GOSUB. ....               | 3                     |
| 6.       | Sub-program. ....         | 5                     |
| 7.       | Name. ....                | 5                     |
| 8.       | Argument. ....            | 6                     |
| 9.       | Parentheses. ....         | 7                     |
| 10.      | Functions. ....           | 7                     |
| 11.      | Intrinsic Functions. .... | 6                     |
| 12.      | User Functions. ....      | 6                     |



# QUESTIONANSWER

 FRAME #  
 FOR REVIEW

| 13. | Function<br>Name | Description |    |
|-----|------------------|-------------|----|
|     | TAB(X)           | 4 .....     | 6  |
|     | SQR(X)           | 7 .....     | 7  |
|     | SIN(X)           | 11 .....    | 9  |
|     | COS(X)           | 13 .....    | 9  |
|     | TAN(X)           | 16 .....    | 9  |
|     | ATN(X)           | 17 .....    | 9  |
|     | LOG(X)           | 1 .....     | 10 |
|     | EXP(X)           | 12 .....    | 10 |
|     | ABS(X)           | 9 .....     | 11 |
|     | SGN(X)           | 15 .....    | 12 |
|     | INT(X)           | 6 .....     | 13 |
|     | RND(X)           | 14 .....    | 14 |
|     | MIN(X1,X2)       | 2 .....     | 15 |
|     | MAX(X1,X2)       | 8 .....     | 15 |
|     | PEEK(X)          | 10 .....    | 16 |
|     | PIN(X)           | 3 .....     | 17 |
|     | POS(X)           | 5 .....     | 18 |
| 14. | DEF. ....        |             | 20 |
| 15. | FN. ....         |             | 20 |
| 16. | Variable. ....   |             | 20 |
| 17. | Real. ....       |             | 21 |
| 18. | Dummy.....       |             | 21 |



# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 6*

### LOOPY DE LOOP

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## Segment 6

# LOOPY DE LOOP

### INTRODUCTION

From what you have learned about BASIC so far, it should be getting apparent that when something can be done to make program writing easier, BASIC will include a feature or Function to do it. The Intrinsic and User Functions that we have just described, the ON. . . GOTO and GOSUB Statements, and the input prompt provisions are all available so that, even though a particular task could be implemented with the "standard" programming tools, it can be done much easier with the "deluxe" features. In this Segment, we will discuss another powerful feature that will reduce programming and typing time. It's known as a "loop", and you will see how BASIC has provided a special Statement pair to help in working with them.



## UNIT OBJECTIVES

When you have completed Segment 6, you will be able to:

1. Name a series of instruction steps that are performed more than once before proceeding with the rest of the program.
2. State if it is possible to construct a loop using "standard" programming Statements.
3. Name a special Statement pair included in BASIC to make it easier to construct loops.
4. Name the three values that must be specified for control of a loop.
5. State the name of the variable which is used to control a loop.
6. State what is used to specify the initial, terminal, and step values of a loop.
7. Name three items which may be included in an expression.
8. Supply the name for a value which is automatically used if no specific step value is specified.
9. State the default step value of a FOR. . . .NEXT loop.
10. Name the Statement which may be used to specify a step value other than 1 for a loop.
11. State if negative values may be used as initial, terminal, and step values of a loop.
12. State if initial, terminal, and step values of a loop may include decimal portions as well as integers.
13. Supply the word that describes a loop which is not completely contained within the body of another loop.
14. State if a FOR. . . .NEXT loop may be "exited" prematurely, such as by using a GOTO Statement.
15. Test your skill at programming using the Statements learned through Segment 6 of this course.

## PROGRAMMED INSTRUCTION

1. In programming, a "loop" is a series of instruction steps that the computer performs more than once before proceeding with the rest of the program. Loops are used often in programming to count, sort, and compare lists of data, make a similar computation on several different data items, and so on.

A series of instruction steps that the computer performs more than once before proceeding with the rest of the program is called a \_\_\_\_\_.

loop

2. We have used several loops in the program examples shown in earlier Segments of this course. Here is a loop, constructed with standard programming Statements, which PRINTs the square root of each number from 10 to 20:

```
10 REM ** LOOP DEMONSTRATION USING STANDARD STATEMENTS **
20 REM
30 REM ESTABLISH STARTING NUMBER
40 LET N=10
50 REM PRINT SQUARE ROOT OF NUMBER
60 PRINT "THE SQUARE ROOT OF";N;"IS";SQR(N)
70 REM ADD ONE TO NUMBER
80 LET N=N+1
90 REM TEST TO SEE IF DONE - IF NOT, LOOP BACK FOR NEXT NUMBER
100 IF N<21 THEN GOTO 60
110 REM WHEN N IS EQUAL TO 21, WE'RE ALL DONE - GOODBYE
120 STOP
```

A loop can be constructed with \_\_\_\_\_ programming Statements.

standard

3. The Statements that actually control the operation of the loop in the above example can be isolated as follows:

```
40 LET N=10
80 LET N=N+1
100 IF N<21 THEN GOTO 60
```

What goes on between lines number 40 and 80 is called the “**body**” of the loop and depends on whatever the particular loop is supposed to accomplish. In Frame 2, the body of the loop PRINTed the square root of the content of variable ‘N’.

The \_\_\_\_\_ of a loop specifies what the loop is to accomplish.

body

4. The program lines in Frame 3 can be called the “**control**” portion of the loop, since they control or determine what the starting value of variable ‘N’ is to be, what value is added to variable ‘N’ each time the loop is performed, and what value of ‘N’ should cause the loop to be “exited” (not performed any more times).

The names of these loop control operations are: **Initial** value, **Step** value and **Terminal** value.

Line 40 of the example program in Frame 2 specifies the \_\_\_\_\_ value of the loop.

initial



5. Suppose we use only the lines of the example program that form the control portion of the loop. For the body of the loop, we will simply PRINT the current value of variable 'N'. Here are the appropriate program lines:

```
40 LET N=10
41 PRINT N;
80 LET N=N+1
100 IF N<21 THEN GOTO 41
110 STOP
```

And a PRINTout:

```
10 11 12 13 14 15 16 17 18 19 20
```

As you can see from the PRINTout, each time we performed the loop the content of variable 'N' was increased by the \_\_\_\_\_ value.

step

6. The program line:

```
100 IF N<21 THEN GOTO 41
```

causes the content of variable 'N' to be tested after it is increased by the step value. As long as the content of variable 'N' is less than 21, the program will loop back to line 41 for another trip through the loop. In this loop, variable 'N' is called the "loop variable".

When the content of variable 'N' reaches 21, the relational test of line 100 comes up "false" and the program will not GOTO line 41. Instead, it proceeds to execute the next line after 100. When this occurs, we say that the loop has been "exited".

In example program line 100, the number 21 is the \_\_\_\_\_ value of the loop.

terminal

7. BASIC includes a special Statement pair to make it easier to construct loops. The Statement pair is

FOR. . . .NEXT

Using the FOR. . . .NEXT Statement pair, the example program in Frame 5 would be written:

```
40 FOR N=10 TO 20
41 PRINT N;
100 NEXT N
110 STOP
```

Lots easier, isn't it?

A PRINTout of the above example would be exactly the same as that shown in Frame 5. The FOR. . . .NEXT Statement pair has simply made it more convenient to construct the loop; let BASIC take care of adding the step value and testing to see if the terminal value has been reached.

Notice, too, that you don't have to figure out the terminal value by adding one more step value than the number of times you wish to perform the loop; if you want 10 to 20, just specify it.

The Statement FOR has as its arguments, the \_\_\_\_\_ value and \_\_\_\_\_ value of the loop.

initial, terminal

8. Suppose you wanted to skip through a loop "by two's". BASIC allows you to specify a particular step value if you wish. If you do not specify a particular step value, BASIC assumes you want it to be 1.

There are several occasions in computer use and programming where you are given the option of selecting a particular value for an operation but, if you do not choose a value, a predetermined figure will be used. When this is the case, the predetermined figure is called a "**default**" value.

The default step value for a FOR. . . .NEXT loop is \_\_\_\_\_.



9. When you wish to specify a step value other than 1 for a FOR...NEXT loop, say 2, you use the Statement STEP. Re-writing our example program of Frame 7 to specify a STEP value of 2 would make it look like this:

```
40 FOR N=10 TO 20 STEP 2
41 PRINT N;
100 NEXT N
110 STOP
```

And a PRINTout:

```
10 12 14 16 18 20
```

To specify a step value other than the default value of 1, the Statement \_\_\_\_\_ is used.

STEP

10. The STEP value in a FOR...NEXT loop can be a negative value if you wish to run through a loop from a high initial value to a low terminal value. Our example loop can be run "backwards" like this:

```
40 FOR N=20 TO 10 STEP -1
41 PRINT N;
100 NEXT N
110 STOP
```

which will PRINT

```
20 19 18 17 16 15 14 13 12 11 10
```

The STEP value can be \_\_\_\_\_ as well as positive.

negative



11. The STEP value does not need to be a whole integer; decimal values are allowed. For example:

```
40 FOR N=10 TO 11 STEP .1
41 PRINT N;
100 NEXT N
110 STOP
```

will PRINT

10 10.1 10.2 10.3 10.4 10.5 10.6 10.7 10.8 10.9 11

STEP values may be \_\_\_\_\_ values as well as whole integers.

decimal

12. The initial and terminal values of a FOR. . .NEXT loop may be either positive or negative values. For example:

```
40 FOR N=-5 TO 5
41 PRINT N;
100 NEXT N
110 STOP
```

will PRINT

-5 -4 -3 -2 -1 0 1 2 3 4 5

The initial and terminal values of a FOR. . .NEXT loop can be \_\_\_\_\_ as well as positive.

negative

13. The initial, step and terminal values that you specify after the FOR Statement, are **expressions** which may be any mix of real numbers, variable names, and computations. Here is how it would look if you assigned loop parameters using variable names:

```
37 LET A=10
38 LET B=20
39 LET C=2
40 FOR N=A TO B STEP C
41 PRINT N;
100 NEXT N
110 STOP
```

This program will PRINT

10 12 14 16 18 20

\_\_\_\_\_ are used to specify the initial, step, and terminal values of a FOR. . . .NEXT loop.

Expressions

14. When a FOR...NEXT loop appears in the body of another FOR...NEXT loop, it is said to be "nested". BASIC allows several loops to be nested, providing the following rule is observed:

**If a FOR...NEXT loop appears inside another FOR...NEXT loop, it must be completely contained within the body of the other loop.**

For example, this is O.K.:

```
10 FOR X=5 TO 7
20 PRINT X;
30 FOR Y=1 TO 3
40 PRINT Y;
50 NEXT Y
60 NEXT X
70 STOP
```

'X' Loop

'Y' Loop

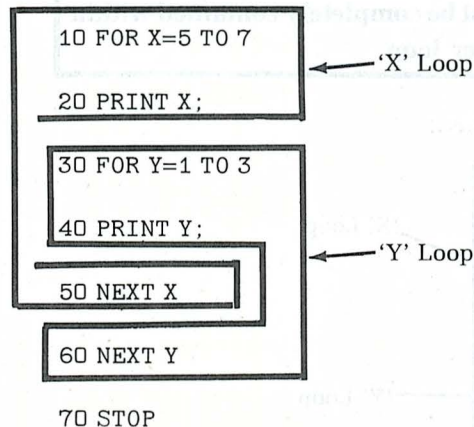
What will the above program PRINT? \_\_\_\_\_

5 1 2 3 6 1 2 3 7 1 2 3



15. FOR . . . NEXT loops may not be “interlaced”. That is, you may not write “NEXT Y” if the immediately preceding “FOR” did not read “FOR Y= . . .”

Here is an example of interlaced loops:



Interlacing of FOR . . . NEXT loops (is) (is not) allowed.

is not

16. In writing a program, it is sometimes desirable to “exit” a FOR . . . NEXT loop prematurely; that is, before the terminal value has been reached. A GOTO Statement may be used to “jump out of a loop”. Here is how a premature exit might be written:

```

10 FOR N=1 TO 10
20 IF 2*N>15 THEN GOTO 40
30 NEXT N
40 PRINT "TWO TIMES";N;"EQUALS";2*N
50 STOP

```

And the PRINTout:

TWO TIMES 8 EQUALS 16

In this example, the relational test in line 20 came “true” before the terminal value of the loop had been reached. When this occurred, the IF . . . THEN Statement caused the program to “jump out of the loop”, ending it.

You may exit a loop prematurely by using a \_\_\_\_\_ Statement to “jump out of the loop”.

GOTO

17. A GOTO Statement may also be used to cause program jumps within the loop. Here is the example of Frame 16 rearranged slightly:

```
10 FOR N=1 TO 10
20 IF 2*N>15 THEN GOTO 40
30 PRINT "TWO TIMES";N;"EQUALS";2*N
40 NEXT N
50 STOP
```

This program will PRINT

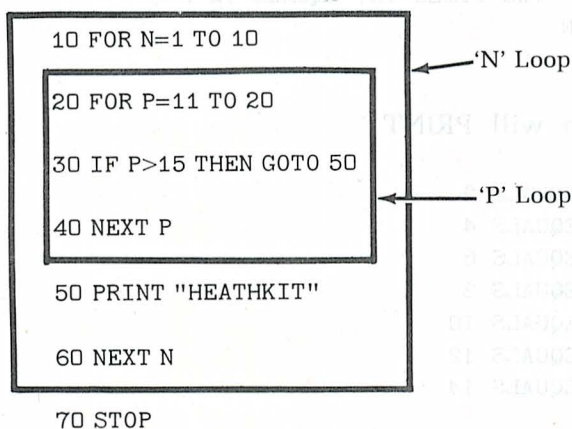
```
TWO TIMES 1 EQUALS 2
TWO TIMES 2 EQUALS 4
TWO TIMES 3 EQUALS 6
TWO TIMES 4 EQUALS 8
TWO TIMES 5 EQUALS 10
TWO TIMES 6 EQUALS 12
TWO TIMES 7 EQUALS 14
```

GOTO Statements (can) (can not) be used to cause program jumps within a loop.

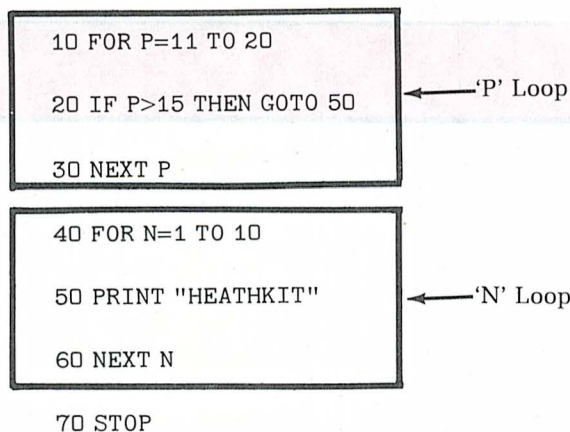
can

18. For the same reason that interlaced loops are not permitted, you cannot use a GOTO Statement to exit one loop prematurely and jump into the **middle** of another loop unless the FOR Statement of the second loop has previously been executed.

This is O.K.:



This is **not** O.K.:



The reason this is not acceptable is that when the relational test of line 20 comes "true" and causes the program to jump to line 50, the Statement "NEXT N" will be encountered before the initial, step and terminal values of the 'N' loop have ever been established. In other words, you can't have a "NEXT N" without having a "FOR N=. . . ." first!

The program examples in this Frame do not include a STEP Statement because the \_\_\_\_\_ step value of 1 was desired.

default



19. The "Stack of Money" program, which we created in Segment 4, used a loop to test the specified denomination of U.S. Currency against a list of "legal" bills. Here are some isolated lines of that program:

```
29 PRINT
30 INPUT "DENOMINATION OF BILL? ";D
32 RESTORE
33 READ A
34 IF D=A THEN GOTO 40
36 IF A<>999 THEN GOTO 33
37 PRINT TAB(5) "THERE IS NO";D;"DOLLAR BILL"
38 GOTO 29
40 INPUT "HEIGHT OF STACK IN FEET? ";H
108 DATA 1,2,5,10,20,50,100,999
```

Do you recognize the loop in these lines?

Knowing how many items there are in our DATA list, we can write a FOR...NEXT loop that will make the same test for a "legal" denomination but will eliminate the "stopper" in the DATA list and the need to test for it.

Here is how the revised program portion would look:

```
29 PRINT
30 INPUT "DENOMINATION OF BILL? ";D
32 RESTORE
33 FOR X=1 TO 7
34 READ A
35 IF D=A THEN GOTO 40
36 NEXT X
37 PRINT TAB(5) "THERE IS NO";D;"DOLLAR BILL"
38 GOTO 29
40 INPUT "HEIGHT OF STACK IN FEET? ";H
108 DATA 1,2,5,10,20,50,100
```

In executing a loop, BASIC must read \_\_\_\_\_  
before it reads "NEXT X".

FOR X= .....

Turn to your Workbook and perform the exercises labeled "Exercise Group 6". Then return to this page.



## REVIEW TEST

1. A series of instruction steps that the computer performs more than once before proceeding with the rest of the program is called a \_\_\_\_\_.
2. It (is) (is not) possible to construct a loop using "standard" programming Statements.
3. BASIC includes the special Statement pair \_\_\_\_\_ to make it easier to construct loops.
4. Whether a loop is written with "standard" programming Statements or the special FOR. . . .NEXT Statement pair, three values must be specified for control of the loop. These values are: \_\_\_\_\_ value, \_\_\_\_\_ value and \_\_\_\_\_ value.
5. The variable assigned to control the loop is called the \_\_\_\_\_.
6. \_\_\_\_\_ are used to specify the initial, terminal, and step values of a FOR. . . .NEXT loop.
7. All expressions, even those used to specify the initial, terminal, and step values of a FOR. . . .NEXT loop, may include any mix of \_\_\_\_\_ numbers, \_\_\_\_\_, and \_\_\_\_\_ names; even other loop variables.
8. If a specific step value is not specified in a FOR. . . .NEXT loop, the \_\_\_\_\_ value of \_\_\_\_\_ is automatically used.
9. If a step value other than 1 is desired for a FOR. . . .NEXT loop, it may be specified by using the Statement \_\_\_\_\_.
10. Initial, terminal, and step values of FOR. . . .NEXT loops (can) (can not) be negative as well as positive.
11. Initial, terminal, and step values of FOR. . . .NEXT loops (can) (can not) include decimal portions as well as integers.
12. Loops may be "nested", meaning that one loop may be completely contained within the body of another loop, but they may not be \_\_\_\_\_, meaning that a loop is not completely contained within the body of another loop.



13. A FOR . . . NEXT loop (can) (can not) be “exited” prematurely, such as by using a GOTO Statement.
14. Test your skill at programming using the Statements learned so far in the course. Write a complete program to produce the following PRINTout of a Multiplication Table. The requirements are:
- When your program is RUN, it is to produce a PRINTout **exactly** as shown below - including the extra spaces which “format” the PRINTout.
  - The program is to **compute** the entries to be PRINTed in the Table.
  - Use a loop to produce the column headings.
  - Use two nested loops to produce the Table entries.
  - REMARK Statements should be included to document and explain the program.

This Test Question is **not** easy. Only a very few will get it 100% right, but give it your best try. You may use your computer to work on the program and you may refer back to the Lesson material.

Objective PRINTout:

MULTIPLICATION TABLE

|   | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
|---|---|----|----|----|----|----|----|----|----|
| 1 | 1 | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  |
| 2 | 2 | 4  | 6  | 8  | 10 | 12 | 14 | 16 | 18 |
| 3 | 3 | 6  | 9  | 12 | 15 | 18 | 21 | 24 | 27 |
| 4 | 4 | 8  | 12 | 16 | 20 | 24 | 28 | 32 | 36 |
| 5 | 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| 6 | 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 |
| 7 | 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 |
| 8 | 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 |
| 9 | 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

## REVIEW ANSWERS

| <u>QUESTION</u> | <u>ANSWER</u>                                                                                                                                                  | <u>FRAME #<br/>FOR REVIEW</u> |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------|
| 1.              | Loop. ....                                                                                                                                                     | 1                             |
| 2.              | Is. ....                                                                                                                                                       | 2                             |
| 3.              | FOR. . . .NEXT. ....                                                                                                                                           | 7                             |
| 4.              | Initial, terminal, step. ....                                                                                                                                  | 4                             |
| 5.              | Loop variable. ....                                                                                                                                            | 6                             |
| 6.              | Expressions. ....                                                                                                                                              | 13                            |
| 7.              | Real, computations, variable. ....                                                                                                                             | 13                            |
| 8.              | Default, 1. ....                                                                                                                                               | 8                             |
| 9.              | STEP. ....                                                                                                                                                     | 9                             |
| 10.             | May. ....                                                                                                                                                      | 10 & 12                       |
| 11.             | May. ....                                                                                                                                                      | 11                            |
| 12.             | Interlaced. ....                                                                                                                                               | 14 & 15                       |
| 13.             | Can ....                                                                                                                                                       | 16                            |
| 14.             | Any program that you wrote which produced an exact duplicate of the Objective PRINTout and met the other requirements specified is "correct." (See Page 6-20.) |                               |

Here is a program which is "correct" according to the definition in answer 14.

```
10 REM ** PROGRAM TO PRINT MULTIPLICATION TABLE **
20 REM
30 REM PRINT HEADING
40 PRINT TAB(17) "MULTIPLICATION TABLE"
50 PRINT
60 REM PRINT COLUMN HEADINGS USING LOOP
70 PRINT " ";
80 FOR H=1 TO 9
90 PRINT " ";H;
100 NEXT H
110 PRINT
120 PRINT
130 REM ESTABLISH MULTIPLIER LOOP
140 FOR X=1 TO 9
150 REM PRINT MULTIPLIER AS ROW HEADING
160 PRINT X;" ";
170 REM ESTABLISH MULTIPLICAND LOOP
180 FOR Y=1 TO 9
190 REM CHECK IF EXTRA SPACE NEEDED
200 IF X*Y<10 THEN PRINT " ";
210 REM COMPUTE AND PRINT RESULT
220 PRINT X*Y;" ";
230 REM LOOP THROUGH ALL MULTIPLICANDS
240 NEXT Y
250 REM BLANK 'PRINT' SETS UP FOR NEXT LINE
260 PRINT
270 REM LOOP THROUGH ALL MULTIPLIERS
280 NEXT X
290 REM ALL DONE - GOODBYE
300 STOP
```



**HEATHKIT  
CONTINUING  
EDUCATION**

# **Individual Learning Program**

## **BASIC PROGRAMMING**

### **SEGMENT 7**

### **LISTS AND ARRAYS**

**EC-1100**

**HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022**

**Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America**

## Segment 7

# LISTS AND ARRAYS

### INTRODUCTION

In this Segment, we are going to make a very important addition to your set of “deluxe” programming tools. As we have demonstrated, BASIC has the ability to keep track of numerical values in a way that allows you to refer to a particular storage location by name. We have called these storage locations “variables” since the numerical contents may change, but the name you have assigned remains the same throughout a program.

So far, we have been limited as to the names we may use in assigning variables to the 26 letters of the alphabet optionally followed by one of the digits 0 through 9. This gives a group of 286 names from which to choose; usually more than enough for the most complex program. We will refer to this group of variables as “**simple**” variables.

The logical mind will immediately perceive that if there are simple variables, there must be “not-so-simple” variables as well. Which, of course, is precisely what we are leading up to.

## UNIT OBJECTIVES

When you have completed Segment 7, you will be able to:

1. Specify the type of variable whose name consists of any legal variable name followed by an expression enclosed in parentheses.
2. Supply the word that describes a group of variables having the names S(1), S(2), S(3), and S(4).
3. State what BASIC must be told before any subscripted variables are assigned in a program.
4. Name the Statement that tells BASIC the size of arrays that will be used in a program.
5. State whether or not a single DIMension Statement can be used to describe the size of more than one array, if required.
6. State the name of the one dimensional array, or list, and specify the number of elements in that list, which the Statement:

```
10 DIM S(10)
```

tells BASIC to reserve space in its memory to accommodate.

7. Write the names of all elements of the list which is DIMensioned by the Statement:

```
10 DIM S(10)
```

8. Supply the name for an array which consists of variables having subscripts with two expressions separated by a comma.
9. Specify the errors in sample programs involving arrays.
10. Supply program lines to correct the errors in sample programs involving arrays.
11. State what a sample program involving arrays will PRINT after the errors have been corrected.



## PROGRAMMED INSTRUCTION

1. As a brief review of what we have learned about the care and feeding of variables, remember that:
  - a. Numerical values are stored by BASIC in named storage locations.
  - b. You choose the name of each storage location from a list of 286 "legal" names.
  - c. Until you assign some other value, the value "zero" is stored as the content of all variables.
  - d. The Statement that names a variable and assigns a value to it is "LET".
  - e. When you wish to use the current content of a variable in a numerical expression, you simply refer to the name of the variable.

As a name for a simple numeric variable, "A7" is (legal) (illegal) while "7A" is (legal) (illegal).

legal, illegal

2. Suppose you were writing a program to compute the average scores and handicaps of golfers at the Green Tee Country Club. One of the requirements would be to enter the scores achieved on each hole of the course during a round of play by a particular golfer.

You might write:

```
10 INPUT "SCORE ON HOLE #1? ";A
20 INPUT "SCORE ON HOLE #2? ";B
30 INPUT "SCORE ON HOLE #3? ";C
```

```
180 INPUT "SCORE ON HOLE #18? ";R
```

What variable would contain the score for the 12th hole?

\_\_\_\_\_

L

3. The question at the end of the last Frame was an easy one to answer but it probably took you a little time to count down the alphabet to determine which letter was in the 12th position. You would have to do a lot of alphabet counting, or make up a chart for reference, if you set out to write a program to keep track of scores for several players simultaneously. Also, it would take a large number of program lines just to enter the scores of several players.

In the last Segment, we discussed program loops and you're probably wondering why we don't use one to obtain the scores for each of the 18 holes. To be sure, FOR S=1 TO 18 etc., would be a much better way to do it, but how are you going to come up with 18 different variables where the scores will be stored?

Glad you asked!

A numeric variable whose name is a single letter or a single letter followed by a single digit is called a \_\_\_\_\_ variable.

simple

4. BASIC offers a different group of names for you to use in assigning “not-so-simple” variables. The variables that you assign using names from this group are infinitely more useful in many cases than the simple variables we have previously discussed. You will remember that everything must have a name, so here is the name used to describe the not-so-simple type of variable; it is called a:

#### Subscripted Variable

The names for subscripted variables begin with any legal name for a simple variable, such as “A” or “A7”. After this name, a number enclosed in parentheses (called a **subscript**) is added. Here are a few legal names for subscripted variables:

A(7)    A7(7)    C(21)    X9(173)    R(1)    R1(1)

BASIC knows the difference between subscripted variables and simple ones. Thus, ‘S’, ‘S3’, and ‘S(3)’ are all legal names for completely different variables.

A legal variable name followed by a number enclosed in parentheses is called a \_\_\_\_\_.

subscripted variable



5. The thing that makes subscripted variables so useful is the fact that the subscript is an **expression**, which may be any mix of real numbers, computations, variable names or function calls.

Suppose, as in our example of entering golf scores, you assign variables S(1) through S(18) to store the scores for the 18 holes of the course. You have selected the name "S" to stand for "scores" and the subscripts 1 through 18 to refer to each hole of the course. The subscripted variables S(1), S(2), etc., through S(18) are said to comprise a "one-dimensional array", or more popularly, a "list".

Each of the subscripted variables in the "S" list is called an "**element**" of that list. For example, the score achieved on the 12th hole may be said to be stored in the 12th element of the "S" list.

An expression enclosed in parentheses and following a variable name is called a \_\_\_\_\_.

subscript

6. Before considering some example programs involving subscripted variables, we must discuss how to tell BASIC that we are going to use an array variable and how many elements it will have.

Each program that includes arrays must provide this information before any assignments are made to subscripted variables so that BASIC can reserve enough room in its memory to store the values. The Statement that tells BASIC to reserve space in its memory for array variable values is:

DIM

which stands for DIMension.

Subscripted variables S(1), S(2), S(3), etc., through S(18) comprise a one-dimensional array or \_\_\_\_\_.

list

7. In the example of storing golf scores, we know that we are going to need 18 elements in the "S" list. To communicate this fact to BASIC, our program will have the following Statement:

```
10 DIM S(18)
```

before we assign a value to any element of the "S" list.

Actually, the Statement DIM S(18) reserves space for 19 elements of the "S" list, since S(0) is a legal element of the list. However, it will be more convenient to refer to each hole of the golf course by its normal number, and the extra element in the list does not have to be used.

The Statement DIM B4(23) reserves space for \_\_\_\_\_ elements of a one-dimensional array or list, whose name is \_\_\_\_\_.

24, B4

8. Now you can write a program that will accept and store the scores one golfer achieved on the 18 holes of the Green Tee Course. You will see how a FOR...NEXT loop is used to control the entry of the scores, how the current content of the loop variable is used as the expression that makes up the subscript of the array variables, and how the DIMension Statement is used to reserve space for the array.

```
10 REM ** PROGRAM TO STORE GOLF SCORES **
20 REM
30 REM TELL BASIC WE'RE USING AN ARRAY AND HOW BIG IT IS
40 DIM S(18)
50 REM SET UP LOOP FOR ENTRY OF 18 SCORES
60 FOR X=1 TO 18
70 REM ASK FOR INPUT OF A SCORE
80 PRINT "SCORE FOR HOLE #";X;
90 REM GET SCORE AND STORE IT IN PROPER ELEMENT OF ARRAY
100 INPUT S(X)
110 REM LOOP FOR NEXT SCORE ENTRY
120 NEXT X
130 REM ALL DONE - GOODBYE
140 STOP
```

Presuming the Statement DIM S(18) has been executed, the subscripted variable S(0) is (legal) (illegal).

legal



9. The contents of subscripted variables can be used in expressions just like the contents of simple variables. For example, suppose that you wanted to PRINT the golf scores obtained by the program in Frame 8 and compute the score for the round. The following program lines could be added to those of Frame 8 to provide a PRINTout. (We have started with line 130, so the STOP Statement is moved to the end of the whole program.)

```
130 REM NOW PRINT THE SCORES AND THE TOTAL SCORE
140 REM FIRST, PRINT HEADING
150 PRINT "HOLE: 1 2 3 4 5 6 7 8 9 10";
160 PRINT " 11 12 13 14 15 16 17 18"
170 PRINT "SCORE: ";
180 REM SET UP LOOP TO PRINT 18 SCORES
190 FOR X=1 TO 18
200 REM PRINT SCORE FOR HOLE
210 PRINT S(X);" ";
220 REM KEEP RUNNING TOTAL
230 LET T=T+S(X)
240 REM LOOP FOR NEXT SCORE
250 NEXT X
260 REM DONE SCORES - PRINT 2 BLANK LINES TO SEPARATE
270 PRINT
280 PRINT
290 REM PRINT TOTAL SCORE
300 PRINT "SCORE FOR ROUND: ";T
310 REM ALL DONE - GOODBYE
320 STOP
```

In the combined program for Frames 8 and 9, the score achieved by the golfer on hole 12 is stored in variable \_\_\_\_\_.

S(12)



10. Regardless of the fact that the example program “formatted” the PRINTout of the golf scores across the page, it is helpful to consider a one-dimensional array as actually being a list of variables in the computer’s memory, and that the list is arranged like this:

S(1)

S(2)

S(3)

S(4)

.

.

.

S(18)

The reason for picturing a one-dimensional array in this manner is that it makes it easier to visualize our next topic; the two-dimensional array or “**matrix**”.

If you were going to write down the scores achieved by several golfers on each hole of the course, and the total of the round for each player, you might do it like this:

HOLE: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 TOT

PLAYER

#1 6 5 5 3 5 5 6 3 5 4 4 5 6 5 6 4 4 3 84

#2 5 7 4 5 5 6 3 5 4 4 4 6 5 5 4 4 6 4 84

#3 4 6 6 5 5 6 4 1 4 5 3 5 5 6 3 5 6 4 83

The chart consists of “**rows**” (each player) and “**columns**” (each hole and the total). Any intersection of player and hole can be specified by a number, such as “2,12”. In the case of “2,12”, we are indicating the score achieved by golfer #2 (thus, row 2) on hole #12 (thus, column 12). At the intersection of row 2 and column 12 is the number “6”. This is the score golfer #2 shot on hole #12.

A two-dimensional array is also called a \_\_\_\_\_.

matrix

11. In the first part of this Segment, we showed how a subscripted variable is specified when we wish to construct a one-dimensional array; S(12) is an example. To construct a two-dimensional array, such as the one described in Frame 10, we can specify a subscripted variable with two subscript arguments; S(2,12) as an example.

The two subscript arguments, since they are separated by a comma, are considered by BASIC to be two independent expressions. This allows you to specify each argument using a different simple variable or any other common method of constructing expressions.

The subscript of a two-dimensional array variable has \_\_\_\_\_ arguments separated by a \_\_\_\_\_.  
Each argument is an independent \_\_\_\_\_.

two, comma, expression

12. A program to store the scores of several golfers, compute the total scores for the round of play, and PRINT the information in the format shown in Frame 10 is a good demonstration of the use of a matrix in BASIC. Points of particular interest in the program are: keyboard INPUT to the 'N' variable of the number of players, which is then used as one argument in the DIMension Statement and to establish the terminal value of a FOR...NEXT loop; use of nested FOR...NEXT loops to control entry and PRINTout of players and scores; and the use of the current contents of loop variables to form the arguments of subscripted variables.

```

10 REM ** TWO-DIMENSIONAL ARRAY DEMONSTRATION PROGRAM **
20 REM
30 REM KEYBOARD INPUT OF NUMBER OF PLAYERS
40 INPUT "HOW MANY PLAYERS? ";N
50 REM TELL BASIC HOW LARGE A MATRIX WE ARE GOING TO USE
60 DIM S(N,19)
70 REM SET UP NESTED LOOPS FOR NUMBER OF PLAYERS
80 REM AND NUMBER OF HOLES AT GOLF COURSE
90 FOR P=1 TO N
100 FOR H=1 TO 18
110 REM KEYBOARD INPUT OF SCORES
120 PRINT "SCORE FOR PLAYER #";P;"ON HOLE";H;
130 INPUT S(P,H)
140 REM LOOP UNTIL DONE HOLES
150 NEXT H
160 REM LOOP UNTIL DONE PLAYERS
170 NEXT P
180 REM PRINT HEADINGS FOR PRINTOUT
190 PRINT " HOLE: 1 2 3 4 5 6 7 8 9 10";
200 PRINT " 11 12 13 14 15 16 17 18 TOT"
210 PRINT "PLAYER"
220 REM SET UP NESTED LOOPS FOR PRINTOUT
230 FOR P=1 TO N
240 PRINT " #";P;" ";
250 FOR H=1 TO 18
260 PRINT S(P,H);
270 REM KEEP RUNNING TOTAL
280 LET T=T+S(P,H)
290 REM LOOP UNTIL DONE HOLES
300 NEXT H
310 REM PRINT TOTAL SCORE FOR ROUND
320 PRINT T
330 REM CLEAR OUT TOTAL SCORE FOR NEXT PLAYER
340 LET T=0
350 REM LOOP UNTIL DONE PLAYERS
360 NEXT P
370 REM ALL DONE - GOODBYE
380 STOP

```

----- cont'd. -----



**12. Cont'd.**

In the Golf Score program, what does line 60 do? \_\_\_\_\_

Reserves space in the computer's memory for a matrix having 'N' rows and 20 columns. (Column '0' is not used in program)

- 13.** A final word about subscripts. It is not allowable to assign a variable having a negative subscript, such as S(-12). When you use an expression to establish the subscript value, take care that it doesn't evaluate to a negative value.

Also note that BASIC will automatically compute the integer of all subscript expressions. Thus, S(3.14159) and S(3) are the same element of the 'S' list.

The variable 'S(ABS(-12))' is (legal) (illegal).

legal

- 14.** And, a final word about the DIMension Statement.

To reduce the amount of typing you need to do and also cut down on the number of program lines, a single DIMension Statement can describe several arrays; as many in fact as you can get on a line of typing. Simply arrange all the array dimensions in a list following the DIMension Statement. Here's how it's done:

```
10 DIM S(15),R(W,Z),P(5),X(20,30),A(D)
```

and so on.

DIMensioning an array with the Statement "DIM R(W,Z)" is legal because the arguments "W" and "Z" are \_\_\_\_\_.

expressions

Turn to your Workbook and perform the exercises labeled "Exercise Group 7". Then return to this page.

1. The first step in the process of identifying a problem is to determine the nature of the problem. This is done by asking the following questions:

- a. What is the problem?
- b. What are the symptoms of the problem?
- c. What are the causes of the problem?
- d. What are the effects of the problem?

2. The second step in the process of identifying a problem is to determine the scope of the problem. This is done by asking the following questions:

- a. How often does the problem occur?
- b. How long does the problem last?
- c. How many people are affected by the problem?
- d. How much money is lost due to the problem?

3. The third step in the process of identifying a problem is to determine the priority of the problem. This is done by asking the following questions:

- a. Is the problem a safety hazard?
- b. Is the problem a health hazard?
- c. Is the problem a financial hazard?
- d. Is the problem a legal hazard?

4. The fourth step in the process of identifying a problem is to determine the responsibility for the problem. This is done by asking the following questions:

- a. Who is responsible for the problem?
- b. What are the responsibilities of each person involved?
- c. How can the responsibilities be assigned?
- d. How can the responsibilities be monitored?

5. The fifth step in the process of identifying a problem is to determine the resources available for the problem. This is done by asking the following questions:

- a. What resources are available?
- b. How can the resources be used?
- c. How can the resources be monitored?
- d. How can the resources be evaluated?

6. The sixth step in the process of identifying a problem is to determine the actions to be taken. This is done by asking the following questions:

- a. What actions are to be taken?
- b. How can the actions be implemented?
- c. How can the actions be monitored?
- d. How can the actions be evaluated?

7. The seventh step in the process of identifying a problem is to determine the results of the actions. This is done by asking the following questions:

- a. What results are to be expected?
- b. How can the results be measured?
- c. How can the results be monitored?
- d. How can the results be evaluated?

8. The eighth step in the process of identifying a problem is to determine the feedback loop. This is done by asking the following questions:

- a. How can the feedback loop be established?
- b. How can the feedback loop be monitored?
- c. How can the feedback loop be evaluated?
- d. How can the feedback loop be improved?

## REVIEW TEST

1. When a legal variable name is followed by an expression enclosed in parentheses, the variable is called a \_\_\_\_\_.
2. A group of variables having the names S(1), S(2), S(3) and S(4) are said to comprise a \_\_\_\_\_ or, more popularly, a \_\_\_\_\_.
3. Before any subscripted variables are assigned in a program, BASIC must be told how large the array will be so that it can \_\_\_\_\_.
4. The Statement that tells BASIC the size of arrays that will be used in a program is \_\_\_\_\_.
5. A single DIMension Statement can describe the size of more than one array, if required. (True) (False)
6. The Statement

10 DIM S(10)

tells BASIC to reserve space in its memory for a one-dimensional array, or list, whose name is \_\_\_\_\_ and which has (how many) \_\_\_\_\_ elements.

7. Write the names of all elements of the 'S' list which is DIMensioned by the Statement in question 6.

8. When the subscript of a subscripted variable has two expressions separated by a comma, the variable is part of a \_\_\_\_\_ or, more popularly, a \_\_\_\_\_.



9. What is wrong with the following program?

```
10 INPUT "HOW MANY PLAYERS? ";N
20 FOR P=1 TO N
30 FOR S=1 TO 10
40 PRINT "PLAYER #";P;"SCORE #";S;
50 INPUT S(P,S)
60 NEXT S
70 NEXT P
80 STOP
```

---

10. Write an appropriate program line that will correct the error in the program of question 9.
- 

11. What is wrong with this program:

```
10 LET S=20
20 LET S3=30
30 LET S(S,S3)=S+S3
40 PRINT S(S,S3)
50 STOP
```

---

12. Write a program line that will correct the error in the program of question 11.
- 

13. After the program in question 11 has been corrected by adding the program line you wrote as the answer to question 12, what will the program PRINT?
-

## REVIEW ANSWERS

| <u>QUESTION</u> | <u>ANSWER</u>                                                                               | <u>FRAME #<br/>FOR REVIEW</u> |
|-----------------|---------------------------------------------------------------------------------------------|-------------------------------|
| 1.              | Subscripted variable. ....                                                                  | 4                             |
| 2.              | One-dimensional array, list. ....                                                           | 5                             |
| 3.              | Reserve space in its memory. ....                                                           | 6                             |
| 4.              | DIM. ....                                                                                   | 6                             |
| 5.              | True. ....                                                                                  | 14                            |
| 6.              | S, 11 ....                                                                                  | 7                             |
| 7.              | S(0) .....<br>S(1)<br>S(2)<br>S(3)<br>S(4)<br>S(5)<br>S(6)<br>S(7)<br>S(8)<br>S(9)<br>S(10) | 6                             |
| 8.              | Two-dimensional array, matrix. ....                                                         | 10                            |
| 9.              | There is no DIMension Statement. ....                                                       | 7                             |
| 10.             | 15 DIM S(N,10) ....                                                                         | 7                             |
| 11.             | There is no DIMension Statement. ....                                                       | 7                             |
| 12.             | 25 DIM S(20,30) ....                                                                        | 7                             |
| 13.             | 50.                                                                                         |                               |







# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 8* STRING SAVERS

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## Segment 8

# STRING SAVERS

### INTRODUCTION

In previous Segments of this course, we have described several ways that BASIC can store, evaluate, and PRINT numbers and computations. We will now consider how BASIC can perform some of these same feats with words as well as numbers.

The Statements to be described are not implemented in all versions of BASIC. Limitations of available computer memory dictate the number of features that can be included. Since the "string capability" we are about to describe requires more than its share of memory, it is usually omitted in less elaborate versions. However, you should learn about these worthwhile programming tools. If you can't use the string Statements in this Segment now, you will be prepared to enjoy them later when you have expanded your system.

### UNIT OBJECTIVES

When you have completed Segment 8, you will be able to:

1. Supply the name for an argument that is a numerical value.
2. Supply the name for an argument that is enclosed in quotation marks.
3. Specify the numeric value of an argument that is a numeric expression.

4. State the numeric value of an argument that is a string literal.
5. Supply the name of a storage location for a numeric value.
6. State the name of a storage location for a string literal.
7. Specify what will be PRINTed by example programs which:
  - A. Sum two numeric variables.
  - B. Concatenate two string variables.
  - C. Make a Relational Test between two string literals.
  - D. Make a Relational Test between the current content of two string variables.
  - E. Store several string literals in a one-dimensional array and then PRINT one element of that array.
  - F. Isolate the leftmost several characters of a string.
  - G. Isolate the rightmost several characters of a string.
  - H. Isolate a group of characters within a string.
  - I. Refer to an ASCII number code.
  - J. Call for the number code of an ASCII character.
  - K. Use a Function to convert string characters to numeric values.
  - L. Use a Function to convert numeric values to string characters.
  - M. Use a Function to measure the length of a string.



## PROGRAMMED INSTRUCTION

1. Look carefully at these two program lines:

```
10 PRINT 1977
20 PRINT "1977"
```

**Both** of them will PRINT:

1977

However, line 10 caused the numerical value “one thousand nine hundred and seventy-seven” to be PRINTed, whereas line 20 caused the four **numbers** 1, 9, 7, and 7 to be PRINTed. Since the result was the same in both cases, you might say, “Picky, picky!” To BASIC, the difference is very important.

The argument in line 20 is called a “string”, while that of line 10 is called a numerical \_\_\_\_\_.

expression

2. When letters, numbers, punctuation marks (and some special characters which we will describe later) are enclosed in quotation marks, a “string” is formed. It is called a string whether it has a single character, as many as the particular version of BASIC being used will permit, or any number in between.

It is particularly important to understand that numbers, when included in a string, have no value. In the program line

```
10 PRINT "76 TROMBONES"
```

the number “76” is simply two PRINTable characters, just like the letters of the word “TROMBONES” are nine PRINTable characters.

In the program line above, the argument “76 TROMBONES” is called a \_\_\_\_\_.

string literal or string

3. In previous Segments, you have seen how numeric values can be stored in BASIC. We do this by naming a variable and assigning the value to it. Because it is a numeric value that is being stored, the named storage location is called a “**numeric variable**”.

Strings, too, may be stored in the same manner as numeric values. Named locations for storing strings are called, not surprisingly, “**string variables**”.

In the program line

```
10 LET A=1977
```

the named storage location “A” is called a \_\_\_\_\_.

numeric variable

4. The rules for selecting names of string variables are the same as those for numeric variables. To mark a variable as one containing a string, the variable name is followed by a **dollar sign**. Examples of legal names for string variables are:

A\$    B7\$    C\$    X9\$

Illegal string variable names are, for example:

AA\$    Because it has two letters  
7B\$    Because it begins with a number  
C22\$    Because it has a two-digit number

Numeric variables and string variables are named in the same way with the exception that the name for a string variable is followed by a \_\_\_\_\_.

dollar sign

5. Just as numeric values are stored in numeric variables, strings are stored in string variables using the Statement "LET". For example:

```
10 LET A$="HEDGEHOG"
```

will store the string "HEDGEHOG" as the **current content** of the string variable "A\$".

When a numeric expression includes the name of a numeric variable, it is the \_\_\_\_\_ of the variable that is used in computing the value of the expression.

current content

6. When the argument of a BASIC Statement includes the **name** of a string variable, it is the current content of the string variable that is used in performing the Statement.

For example:

```
10 LET A$="HEDGEHOG"
20 PRINT A$
30 STOP
```

will PRINT:

HEDGEHOG

When it is desired to use the current content of a string variable in performing a Statement, the \_\_\_\_\_ of the string variable is specified in the argument of the Statement.

name

7. Many of the capabilities BASIC has for dealing with numeric values apply to strings as well. To illustrate, the use of the semicolon and comma in PRINT Statements to **format** the PRINTed output work the same way. For example:

```
10 LET A$="APPLES"
20 LET B$="ORANGES"
30 LET C$="PEARS"
```

----- cont'd. -----



## 7. Cont'd.

```
40 LET D$="BANANAS"
50 PRINT A$;B$;C$;D$
60 PRINT A$,B$,C$,D$
70 STOP
```

will PRINT:

APPLESORANGESPEARSBANANAS

APPLES            ORANGES            PEARS            BANANAS

In the example program, the semicolon tells BASIC to PRINT the list of items on the same line and without adding any spaces, while the comma calls for the items PRINTed to be aligned into \_\_\_\_\_, each 14 characters wide.

columns

8. It's not hard to visualize two numeric values being added together, using the plus sign to indicate the desired operation. Strings can be "added," too, but the result is different.

The word "concatenate" means "to join together in a series." When two or more strings are "added", using the plus sign to indicate the desired operation, the process is called **concatenation**. Here's how it works:

```
10 LET A$="HEDGE"
20 LET B$="HOG"
30 LET C$=A$+B$
40 PRINT C$
50 STOP
```

This program will PRINT:

HEDGEHOG

Remember that when a variable name is specified as part of an expression, it is the current content of the variable that is used in performing the Statement. Because of this, the current content of a string variable can be concatenated with a string literal as in the following example:

```
10 LET A$="HEDGE"
20 PRINT A$+"HOG"
30 STOP
```

cont'd.

This program will PRINT:

HEDGEHOG

When strings are “added” together by use of the plus sign, the process is one of \_\_\_\_\_, meaning “to join together in a series.”

concatenation

In addition to the letters, numbers, and symbols that you can see when they are PRINTed, most keyboards made for use with computers can generate special **control codes** that are used for a variety of purposes. For example, we have mentioned that holding the CTRL key pressed and typing the letter “C” can stop the computer when it is RUNning a BASIC program.

Counting the control codes as well as the PRINTable letters, numbers and symbols, there are 128 different number codes which have been established as the “American Standard Code for Information Interchange”. The code set is generally referred to as “ASCII”.

While your computer may display information in the “Octal” or “Hexadecimal” format, BASIC only knows about decimal numbers. The ASCII number codes, in decimal, are as follows:

ASCII NUMBER CODE CHART (IN DECIMAL)

| Code | Char | Meaning                   | Code | Char   |
|------|------|---------------------------|------|--------|
| 0    | NUL  | Null (nothing;idle)       | 64   | @      |
| 1    | SOH  | Start of Heading          | 65   | A      |
| 2    | STX  | Start of Text             | 66   | B      |
| 3    | ETX  | End of Text               | 67   | C      |
| 4    | EOT  | End of Transmission       | 68   | D      |
| 5    | ENQ  | Enquiry or Who are You?   | 69   | E      |
| 6    | ACK  | Acknowledged              | 70   | F      |
| 7    | BEL  | Bell or Alarm             | 71   | G      |
| 8    | BS   | Backspace                 | 72   | H      |
| 9    | HT   | Horizontal Tabulation     | 73   | I      |
| 10   | LF   | Line Feed                 | 74   | J      |
| 11   | VT   | Vertical Tabulation       | 75   | K      |
| 12   | FF   | Form Feed                 | 76   | L      |
| 13   | CR   | Carriage Return           | 77   | M      |
| 14   | SO   | Shift Out                 | 78   | N      |
| 15   | SI   | Shift In                  | 79   | O      |
| 16   | DLE  | Data Link Escape          | 80   | P      |
| 17   | DC1  | Device Control #1         | 81   | Q      |
| 18   | DC2  | Device Control #2         | 82   | R      |
| 19   | DC3  | Device Control #3         | 83   | S      |
| 20   | DC4  | Device Control #4         | 84   | T      |
| 21   | NAK  | Not Acknowledged          | 85   | U      |
| 22   | SYN  | Synchronize               | 86   | V      |
| 23   | ETB  | End of Transmission Block | 87   | W      |
| 24   | CAN  | Cancel                    | 88   | X      |
| 25   | EM   | End of Medium             | 89   | Y      |
| 26   | SUB  | Substitute                | 90   | Z      |
| 27   | ESC  | Escape                    | 91   | [      |
| 28   | FS   | File Separator            | 92   | \      |
| 29   | GS   | Group Separator           | 93   | ]      |
| 30   | RS   | Record Separator          | 94   | ^      |
| 31   | US   | Unit Separator            | 95   | _      |
| 32   | Sp   | Space                     | 96   | `      |
| 33   | !    |                           | 97   | a      |
| 34   | "    |                           | 98   | b      |
| 35   | #    |                           | 99   | c      |
| 36   | \$   |                           | 100  | d      |
| 37   | %    |                           | 101  | e      |
| 38   | &    |                           | 102  | f      |
| 39   | '    |                           | 103  | g      |
| 40   | (    |                           | 104  | h      |
| 41   | )    |                           | 105  | i      |
| 42   | *    |                           | 106  | j      |
| 43   | +    |                           | 107  | k      |
| 44   | ,    |                           | 108  | l      |
| 45   | -    |                           | 109  | m      |
| 46   | .    |                           | 110  | n      |
| 47   | /    |                           | 111  | o      |
| 48   | 0    |                           | 112  | p      |
| 49   | 1    |                           | 113  | q      |
| 50   | 2    |                           | 114  | r      |
| 51   | 3    |                           | 115  | s      |
| 52   | 4    |                           | 116  | t      |
| 53   | 5    |                           | 117  | u      |
| 54   | 6    |                           | 118  | v      |
| 55   | 7    |                           | 119  | w      |
| 56   | 8    |                           | 120  | x      |
| 57   | 9    |                           | 121  | y      |
| 58   | :    |                           | 122  | z      |
| 59   | ;    |                           | 123  | {      |
| 60   | <    |                           | 124  |        |
| 61   | =    |                           | 125  | }      |
| 62   | >    |                           | 126  | ~      |
| 63   | ?    |                           | 127  | DElete |



9. The ASCII number codes given in the Chart are useful in helping you understand another trick which BASIC can perform with strings: **Relational Tests**.

Remember the Relational Operators? They are:

| <u>SYMBOL</u> | <u>MEANING</u>              |
|---------------|-----------------------------|
| =             | Is equal to                 |
| >             | Is greater than             |
| <             | Is less than                |
| >=            | Is greater than or equal to |
| <=            | Is less than or equal to    |
| <>            | Does not equal              |

What is the outcome of the Relational Test "65<79"? (That's true.)  
(That's false.)

That's true.

10. As you can see from the ASCII Number Code Chart, letters which are nearer the beginning of the alphabet have lower number codes than letters which are "higher" in the alphabet. By using these number codes, BASIC can determine whether a letter is "lower" or "higher" than another and thus make Relational Tests possible.

For example:

```
10 LET A$="A"
20 LET B$="O"
30 IF A$<B$ THEN GOTO 60
40 PRINT A$;" IS GREATER THAN ";B$
50 GOTO 70
60 PRINT A$;" IS LESS THAN ";B$
70 STOP
```

will PRINT:

A IS LESS THAN O

Relational Tests of strings are based on ASCII \_\_\_\_\_.

number codes

11. In Frame 10, we made a Relational Test on two strings, each having a single character. A string composed of a single character is said to have a "Length 1".

Relational Tests of longer strings can be made, too. BASIC simply starts at the beginning of one string and makes the specified test against the other string on a letter-by-letter basis until it runs out of letters in one or the other string. In this manner, BASIC can "alphabetize" lists of string data, such as names. Here's how it works:

```
10 LET A$="APPLES"
20 LET B$="ORANGES"
30 IF A$<B$ THEN GOTO 60
40 PRINT A$;" ARE GREATER THAN ";B$
50 GOTO 70
60 PRINT A$;" ARE LESS THAN ";B$
70 STOP
```

Here's a PRINTout:

APPLES ARE LESS THAN ORANGES

Relational Tests are made on string data on a \_\_\_\_\_-\_\_\_\_\_  
\_\_\_\_\_ basis.

letter-by-letter

12. Lists and arrays can be constructed of string variables just like those discussed in Segment 7 for numeric variables. The dollar sign, denoting string data, is the only difference. Here's a program to demonstrate a string matrix:

```

10 DIM S$(4,2)
20 FOR A=1 TO 4
30 FOR B=1 TO 2
40 READ S$(A,B)
50 NEXT B
60 NEXT A
70 DATA "ALABAMA", "MONTGOMERY", "ALASKA", "JUNEAU"
80 DATA "ARIZONA", "PHOENIX", "ARKANSAS", "LITTLE ROCK"
90 PRINT TAB(4) "STATE CAPITALS"
100 PRINT
110 PRINT "STATE", "CAPITAL"
120 PRINT
130 FOR A=1 TO 4
140 FOR B=1 TO 2
150 PRINT S$(A,B),
160 NEXT B
170 PRINT
180 NEXT A
190 STOP

```

This program will PRINT:

#### STATE CAPITALS

| STATE    | CAPITAL     |
|----------|-------------|
| ALABAMA  | MONTGOMERY  |
| ALASKA   | JUNEAU      |
| ARIZONA  | PHOENIX     |
| ARKANSAS | LITTLE ROCK |

The above program constructs a \_\_\_\_\_ named \_\_\_\_\_ and having \_\_\_\_\_ rows and \_\_\_\_\_ columns.

matrix, S\$, 5, 3



13. The demonstration program in Frame 12 shows the use of quotation marks for the string data in the DATA Statements. When string data is being typed on the keyboard in response to an INPUT Statement, the quotation marks are also required to denote string data. For example:

```
10 INPUT "WHAT IS YOUR NAME? ";N$
20 PRINT "HELLO, ";N$
30 STOP
```

Here's how the program looks when we RUN it. (The underlined parts are those typed on the keyboard.)

```
WHAT IS YOUR NAME? "WILLARD"
HELLO, WILLARD
```

Data typed on the keyboard in response to an INPUT Statement, must be surrounded by \_\_\_\_\_.

quotation marks

NOTE: Some versions of BASIC do not require INPUT string data to be quoted. Check your user's manual.

14. If you wish to have string data entered on the keyboard during a program, and do not want to use quotation marks, a special Statement can be used: **Line Input**. Here's how it works:

```
10 LINE INPUT "WHAT IS YOUR NAME? ";N$
20 PRINT "HELLO, ";N$
30 STOP
```

Here is a RUN:

```
WHAT IS YOUR NAME? DOLLY
HELLO, DOLLY
```

Data typed on the keyboard in response to a **Line Input** Statement (does) (does not) have to be surrounded by quotation marks.

does not

15. In Segment 5, we discussed “built-in” subprograms which perform several often-used mathematical computations. Since they deal with numeric values, the subprograms described in Segment 5 are called “Numeric Functions”.

Intrinsic Functions included in BASIC to make it possible for the programmer to manipulate string data are called “**String Functions**”. Let’s see how they work!

FUNCTIONS: CHR\$(X)  
              ASC(X\$)

These two complementary Functions convert number codes to the matching ASCII character and vice-versa. (The number codes for ASCII characters are given in a Chart following Frame 8.)

From the Chart, we see that the number code for the letter “H” is 72. The following program will PRINT the letter “H” when its number code is given as the argument to the Function:

```
10 PRINT CHR$(72)
20 STOP
```

Here’s the PRINTout:

H

While SQR(X) is called a Numeric Function, CHR\$(X) is called a \_\_\_\_\_ Function.

String

16. Conversion from an ASCII character to its number code is obtained with the Function ASC(X\$). For example:

```
10 PRINT ASC("H")
20 STOP
```

will PRINT:

72

If the argument to the Function ASC(X\$) is a string of length greater than 1, conversion is made of the first character of the string only. For example:

```
10 LET A$="HEDGEHOG"
20 PRINT ASC(A$)
30 STOP
```

will PRINT:

72

The Function ASC(X\$) converts an ASCII character to its

number code

17. It's understandable that you might wonder what use can be made of the ASCII number codes that would require the conversions provided by the CHR\$(X) and ASC(X\$) Functions. Let's consider a couple of occasions where they might be used.

BASIC considers the control codes represented by the number codes 0 through 31 as "unPRINTable", and thus rejects them if you type them on your keyboard. Suppose your CRT display will erase the screen if it receives the number code 26. Holding the CTRL key depressed while typing the "Z" key will generate code 26, but BASIC will not allow you to include "CTRL/Z" as part of a string in a program.

So, what do you do if you are writing a program and want to erase the screen from time to time? Well, PRINT CHR\$(26) will tell BASIC to output that number code whether it thinks it is PRINTable or not!

----- cont'd. -----



**17. Cont'd.**

As another example, some printing machines used with computers will recognize the number code 12 as a "Form Feed" command, which causes the paper to advance to the top of the form. Again, BASIC would reject the CTRL/L which generates number code 12. However, you can include the Statement PRINT CHR\$(12) in your program to obtain the Form Feed command to your printer.

The Function CHR\$(X) converts a \_\_\_\_\_ to its equivalent ASCII character.

number code

**18. FUNCTIONS: LEFT \$(X\$,X)  
RIGHT \$(X\$,X)**

The first or last group of characters in a string can be **isolated** with these two Functions. The first argument in the list specifies the string containing the characters to be extracted and the second argument specifies the number of characters to be extracted. Here are examples of the use of these Functions:

```
10 LET A$="HEDGEHOG"
20 PRINT LEFT$(A$,5)
30 STOP
```

This program will PRINT:

HEDGE

Another example:

```
10 LET A$="HEDGEHOG"
20 LET B$=RIGHT$(A$,3)
30 PRINT B$
40 STOP
```

This program will PRINT:

HOG

cont'd.

## 18. Cont'd.

And, finally:

```
10 LET A$="HEDGEHOG"
20 LET B$=LEFT$(A$,5)+RIGHT$(A$,3)
30 PRINT B$
40 STOP
```

This program will PRINT: \_\_\_\_\_

HEDGEHOG

## 19. FUNCTION: MID\$(X\$,X,Y)

A group of characters of any length can be isolated from **within** a string with the MID\$ Function. The arguments in the list specify as follows:

X\$=The string from which characters are to be isolated.

X =The position of the first character of the group to be isolated.

Y =The number of characters to be isolated.

Here's the ubiquitous example:

```
10 LET A$="HEDGEHOG"
20 PRINT MID$(A$,3,5)
30 STOP
```

This program will PRINT:

DGEHO

If the third argument is omitted, the isolated group will begin with the specified character and continue to the end of the specified string. For example:

```
10 LET A$="HEDGEHOG"
20 PRINT MID$(A$,6)
30 STOP
```

This program will PRINT \_\_\_\_\_.

HOG

20. FUNCTIONS: STR\$(X)  
VAL(X\$)

These two Functions implement conversion between numeric values and strings. STR\$ creates a **string** of number characters that represent the numeric argument "X". The Function is useful where the current contents of a numeric variable are converted to a string which can then be incorporated with other string data by concatenation. Here is an example of the use of STR\$:

```
10 LET A=2
20 LET B=38
30 LET C$=STR$(A*B)+"TROMBONES"
40 PRINT C$
50 STOP
```

The program will PRINT:

76 TROMBONES

When an argument to a string Function is a numeric expression, it can be any mix of real numbers, variable names, computations, or

### Numeric Functions

21. If a string contains number characters which you wish to use in numeric computations, the characters can be converted to their **numeric value** with the Function VAL.

For example:

```
10 LET A$="76 TROMBONES"
20 LET B=VAL(LEFT$(A$,2))
30 PRINT "HALF OF THE TROMBONES EQUAL";B/2
40 STOP
```

This program will PRINT:

HALF OF THE TROMBONES EQUAL 38

----- cont'd. -----



**21. Cont'd.**

To explain the Statement in line 20:

Remember that arguments to functions can be other Functions.

The argument to VAL is LEFT\$.

The argument to LEFT\$ is A\$.

The second part of the argument to LEFT\$ is the number of characters from the A\$ string that are to be isolated.

The Function "VAL" converts numeric string characters to a numeric \_\_\_\_\_.

value

**22. FUNCTION: LEN(X\$)**

It is often necessary to determine the number of characters, or "length" of the current content of a string variable. The Function LEN will supply this information for the string variable given as the argument to the Function. The following program illustrates one use of this Function:

```
10 LINE INPUT "TYPE SOMETHING: ";A$
20 PRINT "BACKWARDS, THAT SPELLS:"
30 FOR I=LEN(A$) TO 1 STEP -1
40 PRINT MID$(A$,I,1);
50 NEXT I
60 STOP
```

Here's a PRINTout:

```
TYPE SOMETHING: WILLARD
BACKWARDS, THAT SPELLS:
DRALLIW
```

The Function LEN(X\$) returns the length of the string that is the \_\_\_\_\_ of the string variable X\$.

current content

Let's consider how the LEN(X\$) Function makes the program in the last Frame possible.

In order to PRINT the string backwards, we obviously must start at the last letter. Since the keyboard INPUT could have been any number of characters from one to 80 or more, how can we determine the position number of that letter?

The LEN(X\$) Function returns the number of characters in the string. If there are seven characters, as there are in the example PRINTout, then LEN(A\$)=7, and 7 is also the position number of the last character in the string.

By writing the FOR....NEXT loop to have an Initial value of LEN(A\$), the loop will begin with the position number of the last character as the value of the loop variable. With a Terminal value of 1 and a Step value of -1, the loop will count backwards from the last character position to the first.

Using the loop variable as one of the arguments to the MID\$ Function isolates each of the letters so that they can be PRINTed individually.

### 23. FUNCTION: SPC(X)

This Function will PRINT the number of **spaces** given as the argument to the Function. For example:

```
10 PRINT SPC(20);"HEDGE";SPC(10);"HOG"
20 STOP
```

will PRINT:

HEDGE                      HOG

Be careful to note the difference between the Functions TAB(X) and SPC(X). The argument to TAB specifies an **exact** position on the PRINTed line while SPC specifies the **number of spaces** from the present position on the line. Both are useful in certain situations.

The argument to SPC(X) is a \_\_\_\_\_.

numeric expression

Turn to your Workbook and perform the exercises labeled "Exercise Group 8" then return to this page.





## REVIEW TEST

1. In the program line

```
10 PRINT 1977
```

the argument is a \_\_\_\_\_.

2. In the program line

```
10 PRINT "1977"
```

the argument is a \_\_\_\_\_.

3. In the program line

```
10 PRINT 1977
```

the argument has the numeric value \_\_\_\_\_.

4. In the program line

```
10 PRINT "1977"
```

the argument has the numeric value \_\_\_\_\_.

5. In the program line

```
10 LET A=1977
```

the named storage location is called a \_\_\_\_\_.

6. In the program line

```
10 LET A$="1977"
```

the named storage location is called a \_\_\_\_\_.

7. Consider the following program lines:

```
10 LET A=1977
20 LET B=2
30 LET C=A+B
40 PRINT C
50 STOP
```

What will be PRINTed? \_\_\_\_\_.

8. Consider the following program lines:

```
10 LET A$="1977"
20 LET B$="2"
30 LET C$=A$+B$
40 PRINT C$
50 STOP
```

What will be PRINTed? \_\_\_\_\_.

9. What is the word that describes the joining together of the two strings in the last question? \_\_\_\_\_.

10. Consider the following program lines:

```
10 IF "THOUSAND">"HUNDRED" THEN PRINT "GREATER"
20 IF "THOUSAND"<"HUNDRED" THEN PRINT "LESSER"
30 STOP
```

What will be PRINTed? \_\_\_\_\_.

11. Consider the following program lines:

```
10 LET A$="TEN"
20 LET B$="TWENTY"
30 IF A$>B$ THEN PRINT "GREATER"
40 IF A$<B$ THEN PRINT "LESSER"
50 STOP
```

What will be PRINTed? \_\_\_\_\_.

12. Consider the following program lines:

```
10 DIM R$(9)
20 FOR I=1 TO 9
30 READ R$(I)
40 NEXT I
50 PRINT R$(6)
60 STOP
70 DATA "DASHER", "DANCER", "PRANCER", "VIXEN", "COMET"
80 DATA "CUPID", "DONNER", "BLITZEN", "RALPH"
```

What will be PRINTed? \_\_\_\_\_.



13. Consider the following program lines:

```
10 LET A$="DASHERDANCERPRANCERVIXENCOMETCUPIDDONNERBLITZEN"
20 PRINT LEFT$(A$,6)
30 STOP
```

What will be PRINTed? \_\_\_\_\_.

14. Consider the following program lines:

```
10 LET A$="DASHERDANCERPRANCERVIXENCOMETCUPIDDONNERBLITZEN"
20 PRINT RIGHT$(A$,7)
30 STOP
```

What will be PRINTed? \_\_\_\_\_.

15. Consider the following program lines:

```
10 LET A$="DASHERDANCERPRANCERVIXENCOMETCUPIDDONNERBLITZEN"
20 PRINT MID$(A$,30,5)
30 STOP
```

What will be PRINTed? \_\_\_\_\_.

16. Consider the following program lines:

```
10 PRINT CHR$(88)
20 STOP
```

Referring to the ASCII Number Code Chart following Frame 8,  
what will be PRINTed? \_\_\_\_\_.

17. Consider the following program lines:

```
10 PRINT ASC("X")
20 STOP
```

What will be PRINTed? \_\_\_\_\_.

18. Consider the following program lines:

```
10 LET A$="76 TROMBONES"
20 LET B$="101 CORNETS"
30 PRINT VAL(LEFT$(A$,2))+VAL(LEFT$(B$,3))
40 STOP
```

What will be PRINTed? \_\_\_\_\_.

19. Consider the following program lines:

```
10 LET A=76
20 LET B=101
30 LET C$="THERE WAS A TOTAL OF"+STR$(A+B)+"TROMBONES "
40 LET C$=C$+"AND CORNETS"
50 PRINT C$
60 STOP
```

What will be PRINTed? \_\_\_\_\_.

20. Consider the following program lines:

```
10 LET A$="TIME FOR A COFFEE BREAK"
20 PRINT LEN(A$)
30 STOP
```

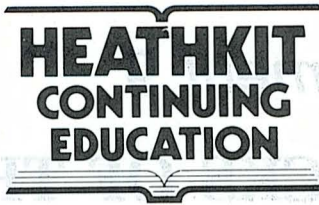
What will be PRINTed? \_\_\_\_\_.

## REVIEW ANSWERS

| QUESTION | ANSWER                                          | FRAME #<br>FOR REVIEW |
|----------|-------------------------------------------------|-----------------------|
| 1.       | Numeric expression. ....                        | 1                     |
| 2.       | String literal. ....                            | 2                     |
| 3.       | 1977. ....                                      | 1                     |
| 4.       | Zero or nothing. ....                           | 2                     |
| 5.       | Numeric variable. ....                          | 3                     |
| 6.       | String variable. ....                           | 3                     |
| 7.       | 1979. ....                                      | 1                     |
| 8.       | 19772. ....                                     | 8                     |
| 9.       | Concatenation. ....                             | 8                     |
| 10.      | GREATER. ....                                   | 9, 10, 11             |
| 11.      | LESSER. ....                                    | 9, 10, 11             |
| 12.      | CUPID. ....                                     | 12                    |
| 13.      | DASHER. ....                                    | 18                    |
| 14.      | BLITZEN. ....                                   | 18                    |
| 15.      | CUPID. ....                                     | 19                    |
| 16.      | X. ....                                         | 15                    |
| 17.      | 88. ....                                        | 15                    |
| 18.      | 177. ....                                       | 20                    |
| 19.      | THERE WAS A TOTAL OF 177 TROMBONES AND CORNETS. | 20                    |
| 20.      | 23. ....                                        | 22                    |







# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 9*

### TRICKS OF THE TRADE

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## Segment 9

# TRICKS OF THE TRADE

### INTRODUCTION

In this final Segment about the “tools” of the BASIC programming language, we have both good news and bad news for you! The good news is that there are some ways to cut down on typing time and reduce memory usage for your programs. The bad news is that there still a few Statements to learn.

First, the good news. Now that you have progressed to your present level of expertise, we can introduce a few exceptions that will make programming easier and which you can take in stride.



## UNIT OBJECTIVES

When you have completed Segment 9, you will be able to:

1. State a condition of use of the keyword LET.
2. Describe a restriction on the omission of the keyword LET.
3. Name the conditional Statement which allows the keyword THEN to be omitted in a Relational Test Statement.
4. State if more than one Statement may be typed on the same program line.
5. Name the character that is used to mark the end of each Statement (except the last one) which appears on the same program line.
6. Identify which Statement on a program line may be used as a "target" for GOTO and GOSUB Statements.
7. Describe the condition that would prevent a Statement from being executed when it appears on the same program line as a Relational Test.
8. State what happens to the portions of a program line that follow a Relational Test when that Test comes up false.
9. State if DATA Statements may appear on the same program line as other Statements.
10. Specify how BASIC switches from conventional to scientific notation.
11. Describe the significance of the letter "E" in scientific notation.
12. State what the numbers following the letter "E" in scientific notation specify.
13. State the direction in which the decimal is moved to convert scientific notation to conventional notation when the sign of the exponent is "+" and when it is "-".
14. List the Boolean Operators implemented in BASIC.

2. Another optional Statement is THEN, but **only** when it is followed by GOTO. For example, see how line 30 in the following program is written:

```
10 A$="76 TROMBONES"
20 B$="101 CORNETS"
30 IF VAL(LEFT$(A$,2))<VAL(LEFT$(B$,3)) GOTO 60
40 PRINT "THERE ARE MORE TROMBONES THAN CORNETS"
50 GOTO 70
60 PRINT "THERE ARE MORE CORNETS THAN TROMBONES"
70 STOP
```

The program will PRINT

THERE ARE MORE CORNETS THAN TROMBONES

In a Relational Statement, THEN is optional only if it is followed by the Statement \_\_\_\_\_.

GOTO

3. You can shorten programs considerably, make them more readable, and conserve memory when you type more than one Statement on a numbered program line. When you do this, be sure that you end each complete Statement (except the last one) with a **colon**.

In this program, lines 10 and 30 have two separate Statements on the same line:

```
10 A$="76 TROMBONES":B$="101 CORNETS"
20 IF VAL(LEFT$(A$,2))<VAL(LEFT$(B$,3)) GOTO 40
30 PRINT "THERE ARE MORE TROMBONES THAN CORNETS":GOTO 50
40 PRINT "THERE ARE MORE CORNETS THAN TROMBONES"
50 STOP
```

Here's the PRINTout:

THERE ARE MORE CORNETS THAN TROMBONES

More than one Statement may be typed on a line if the end of each Statement (except the last one) is marked by a \_\_\_\_\_.

colon



4. There are a couple of things to be especially careful about when you type multiple Statements per line. One is that you can use only the first Statement on the line as a "target" for GOTO and GOSUB Statements, since these specify **line numbers** and not positions on the line.

This program will not work:

```
10 A$="76 TROMBONES":B$="101 CORNETS"
20 IF VAL(LEFT$(A$,2))<VAL(LEFT$(B$,3)) GOTO 40
30 PRINT "THERE ARE MORE TROMBONES THAN CORNETS"
40 GOTO 50:PRINT "THERE ARE MORE CORNETS THAN TROMBONES"
50 STOP
```

because the intended target for the GOTO Statement in line 20 is not the \_\_\_\_\_ Statement on line 40.

first

5. Both a blessing and a curse is the fact that if you type a Statement at the end of the same line as a Relational Test, the Statement will not be executed if the Relational Test comes up "false". While this is a definite programming advantage in some cases, there will be times when you forget this fact and will wonder why your program doesn't work the way it should.

This program will work:

```
10 A$="76 TROMBONES":B$="101 CORNETS"
20 A=VAL(LEFT$(A$,2)):B=VAL(LEFT$(B$,3))
30 IF A<B THEN PRINT "MORE CORNETS THAN TROMBONES":GOTO 50
40 PRINT "MORE TROMBONES THAN CORNETS"
50 STOP
```

The Statement GOTO 50 at the end of line 30 was executed because the Relational Test  $A < B$  came up true.

This program will not work:

```
10 A$="76 TROMBONES":B$="101 CORNETS"
20 A=VAL(LEFT$(A$,2)):B=VAL(LEFT$(B$,3))
30 IF A>B THEN PRINT "TROMBONES":IF A<B THEN PRINT "CORNETS"
40 PRINT "WHO CARES ABOUT TROMBONES AND CORNETS, ANYWAY?"
50 STOP
```

---cont'd.---



## 5. cont'd.

The Relational Test IF  $A < B$  will never be made because the Relational Test IF  $A > B$  came up false and the rest of the line was ignored!

The programmer must be careful in placing additional Statements at the end of lines containing \_\_\_\_\_.

## Relational Tests

6. A final restriction on the use of multiple Statements per line states that a program line on which a **DATA** Statement appears cannot have any other Statement on that line. For this reason, line 20 in the following program is not legal:

```
10 READ A
20 PRINT A; "TROMBONES": DATA 76
30 STOP
```

A program line may have several Statements, separated by colons, providing none of the Statements are \_\_\_\_\_.

## DATA

7. Scientific notation is the term used to describe a method of representing very large and very small numeric values without having to count endless zero's. BASIC **automatically** switches to scientific notation when numeric values reach certain limits.

Here is a numeric value in scientific notation:

3.14159E+6

-----cont'd.-----

**7. cont'd.**

The letter "E" in the example specifies that the next digits are to be taken as the "Exponent" of the value 10. The value before the letter "E" is then multiplied by ten to the specified power.

Simply stated, the digits following the letter "E" denote the number of positions that the **decimal** point is to be moved to represent the value in standard notation. Thus:

| <u>SCIENTIFIC NOTATION</u> | <u>STANDARD NOTATION</u> |
|----------------------------|--------------------------|
| 3.14159E+6                 | 3141590                  |
| 3.14159E+10                | 31415900000              |
| 3.14159E+15                | 3141590000000000         |

and so on.

If the sign of the Exponent is "+" or is omitted, the decimal moves to the **right**. If the sign of the Exponent is "-", the decimal is moved to the **left**. Thus:

| <u>SCIENTIFIC NOTATION</u> | <u>STANDARD NOTATION</u> |
|----------------------------|--------------------------|
| 3.14159E+6                 | 3141590                  |
| 3.14159E6                  | 3141590                  |
| 3.14159E06                 | 3141590                  |
| 3.14159E+06                | 3141590                  |
| 3.14159E-6                 | 0.00000314159            |
| 3.14159E-06                | 0.00000314159            |
| 3.14159E-10                | 0.000000000314159        |

In scientific notation, the value of the Exponent specifies the number of positions that the \_\_\_\_\_ is to be moved.

decimal

8. In scientific notation, the sign of the Exponent tells us the **direction** in which the decimal is to be moved; to the right when the sign is "+", and to the left when the sign is "-". Be careful to note that the sign of the Exponent is completely different from the sign of the numeric value, itself. To illustrate:

SCIENTIFIC NOTATION
STANDARD NOTATION

|              |                |
|--------------|----------------|
| 3.14159E+06  | 3141590        |
| -3.14159E+06 | -3141590       |
| 3.14159E-06  | 0.00000314159  |
| -3.14159E-06 | -0.00000314159 |

In scientific notation, the sign of the Exponent specifies the \_\_\_\_\_ in which the decimal point is to be moved.

direction



9. Although it may vary in different versions, BASIC usually switches to scientific notation for values of 1000000 and larger, and for values of .001 and smaller. You can check this out with the following program:

```
10 REM ** PROGRAM TO CHECK SCIENTIFIC NOTATION **
20 INPUT "NUMBER, PLEASE? ";N
30 PRINT "IN SCIENTIFIC NOTATION, YOUR NUMBER IS: ";N
40 PRINT
50 IF N<>0 GOTO 20
60 STOP
```

Here's the PRINTout when we ran the program:

```
NUMBER, PLEASE? 999999
IN SCIENTIFIC NOTATION, YOUR NUMBER IS: 999999
```

```
NUMBER, PLEASE? 1000000
IN SCIENTIFIC NOTATION, YOUR NUMBER IS: 1.000000E+06
```

```
NUMBER, PLEASE? .01
IN SCIENTIFIC NOTATION, YOUR NUMBER IS: .01
```

```
NUMBER, PLEASE? .001
IN SCIENTIFIC NOTATION, YOUR NUMBER IS: 1.000000E-03
```

```
NUMBER, PLEASE? 0
IN SCIENTIFIC NOTATION, YOUR NUMBER IS: 0
```

BASIC \_\_\_\_\_ switches to scientific notation to represent very large and very small values.

automatically

In addition to decimal values and scientific notation, BASIC knows another form of number representation. It's called:

### BOOLEAN VALUES

A complete discussion of the binary numbering system would be so long that it would distract our attention from the purpose of this course; that of learning the BASIC programming language. An abbreviated discussion would not impart sufficient knowledge of the subject. Therefore, if you are not fully familiar with binary numbers, we suggest that you study the instructional unit we have included as "Appendix B." This unit explains the binary numbering system and some other forms of number notation that are in popular use.

If you need a "refresher", type in and RUN the topic program for the "Experiments" in "Exercise Group 9" in your Workbook (Page 49). This program includes a decimal value to binary conversion feature to get you back on track.

The Boolean Operators implemented in BASIC are:

AND  
OR  
NOT

We will discuss each of these in detail; but first let's consider the **common** characteristics of their use.

10. A typical expression including a Boolean Operator might take the form:

50 AND 26

but 50 AND 26 do **not** equal 76!

Here's an example program:

```
10 A=50 : B=26
20 C=A AND B
30 PRINT C
40 STOP
```

-----cont'd.-----

## 10. cont'd.

This program will PRINT

18

Isn't that a strange result for any conceivable mathematical operation on the two values 50 and 26? (Yes, it is!) (No, it's logical!)

either answer is correct

11. The answer, 18, for the statement 50 AND 26 is strange if a mathematical operation had been specified. The answer is perfectly suitable when you understand that the statement 50 AND 26 calls for a **logical** operation on the binary representation of the two values 50 and 26.

Here's how it works:

|                                  |        |
|----------------------------------|--------|
| Binary representation of 50:     | 110010 |
| Binary representation of 26:     | 011010 |
| Applying logical AND results in: | 010010 |
| Converted to decimal equals:     | 18     |

The Boolean Operator AND specifies a \_\_\_\_\_ operation on the binary representations of two arguments.

logical

The fact to learn from Frames 10 and 11 is that, when one of the Boolean Operators is specified, the argument or arguments are converted to binary representation, the specified logical operation is performed on the binary bits, and the result is converted back to a decimal value.



12. When a Boolean Operation is specified, the decimal values of the argument are converted to **16-bit binary** representation. This limits the range of decimal values which can be used for Boolean Operations to 0 through 65535. Upon conversion to binary, any fractional part of the decimal number is discarded - that is, the decimal value is first converted to an integer value. If the value is negative, or the integer is greater than 65535, an error message will result.

Thus:

| <u>DECIMAL VALUE</u> | <u>BINARY REPRESENTATION</u> |
|----------------------|------------------------------|
| 3.14159              | 0000000000000011             |
| 65535                | 1111111111111111             |
| -3.14159             | Error                        |
| 65536                | Error                        |
| 1977                 | 0000011110111001             |

When a Boolean Operation is specified, decimal values are converted to \_\_\_\_\_ bit binary representation.

16

13. Boolean Operator: AND

The Boolean AND logical operation conforms to the following "truth" equivalences:

0   1   0   1

0   0   1   1

0   0   0   1   Logical AND operation

What is the outcome of 3 AND 2? \_\_\_\_\_

2

# 14. Boolean Operator: OR

The Boolean OR logical operation conforms to the following "truth" equivalences:

|          |          |          |          |                      |
|----------|----------|----------|----------|----------------------|
| 0        | 1        | 0        | 1        |                      |
| <u>0</u> | <u>0</u> | <u>1</u> | <u>1</u> |                      |
| 0        | 1        | 1        | 1        | Logical OR operation |

What is the outcome of 2 OR 1? \_\_\_\_\_

3

# 15. Boolean Operator: NOT

The Boolean NOT is called a unary operator, since it performs its logical operation on a single value. The NOT function simply **complements** the bits of the binary representation; all "0's" become "1's" and all "1's" become "0's".

For example:

```
10 PRINT NOT 65534
20 STOP
```

will PRINT:

1

because:

|                            |                  |
|----------------------------|------------------|
| 65534 in 16-bit binary is: | 1111111111111110 |
| Complementing all bits:    | 0000000000000001 |
| Which in decimal is:       | 1                |

-----.cont'd.-----

## 15. cont'd.

Another example:

```
10 A=NOT 0
20 PRINT A
30 STOP
```

will PRINT:

65535

because:

|                         |                  |
|-------------------------|------------------|
| 0 in 16-bit binary is:  | 0000000000000000 |
| Complementing all bits: | 1111111111111111 |
| Which in decimal is:    | 65535            |

What is the result of the Boolean Operation NOT 65535? \_\_\_\_\_

0

The word “**precedence**” is used to describe the order in which BASIC performs mathematical operations when more than one is specified as part of an expression. Some operations are said to have equal precedence, and BASIC performs these in the order that you list them. For example, in the expression

$$6+4-1+10-3$$

the result is 16, because “+” and “-” have equal precedence.

But, in the expression

$$6+4-1+5*2-3$$

the result will also be 16, instead of 25 as you might expect. This occurs because “\*” has a higher precedence than “+” and “-”; therefore, the operation  $5*2$  was performed first.



In an earlier Segment, we described how you can use parentheses to tell BASIC to make **subcomputations**, and use the **result** in the expression. Thus, for the last example, the expression

$$(6+4-1+5)*2-3$$

will produce the expected answer of 25.

The following chart describes the “order” of precedence for BASIC’s mathematical and Boolean operations:

| CHART OF PRECEDENCE FOR MATH AND BOOLEAN OPERATIONS |                |                |
|-----------------------------------------------------|----------------|----------------|
| Done first                                          | NOT            | Boolean        |
| Done next                                           | ↑              | Exponentiation |
| Done next                                           | * /            | Mathematical   |
| Done next                                           | + -            | Mathematical   |
| Done next                                           | < <= = <> >= > | Relational     |
| Done next                                           | OR             | Boolean        |
| Done last                                           | AND            | Boolean        |

(Operations appearing on the same line in the chart have equal precedence.)

It is particularly important to take precedence into account when you make Relational Tests on expressions containing Boolean Operators.

Consider this program:

```
10 IF 50 AND 26=18 THEN PRINT "YES"
20 STOP
```

In Frame 10, you learned that the expression 50 AND 26 **does** equal 18, but the program above will **not** PRINT “YES”. Why not? The answer to that question lies in the order of precedence of operations.

As the chart shows, “=” has a **higher** precedence than “AND”. Therefore, the Statement first tests to see if 50=18. Since it does not, execution of the program proceeds to the next line, **ignoring** any further portions of line 10.

The solution, of course, is to parenthesize the expression 50 AND 26 to tell BASIC to make that **subcomputation** first; then apply the Relational Test.

Finally, a word about the “up arrow” symbol ( $\uparrow$ ) shown in the Chart. You probably know that the expression

$$4*4*4$$

can also be represented as “four raised to the third power” or “four cubed”, and noted as

$$4^3$$

In BASIC, you can raise a number to any desired power by using the “up arrow” to mark **exponentiation**. Thus, “four cubed” is typed as:

$$4\uparrow 3$$

Both the number and the exponent are **expressions**. Thus,  $A\uparrow B$  is legal!

**16.** There will be some occasions when you would find it handy to be able to clear out the current contents of all program variables. One way is to LET A=0:B=0:C=0 and so on through the list of all the variables that have been assigned in the program.

A far simpler way is by use of the Statement CLEAR. Here is an example program to demonstrate how it works:

```
10 A=76:B=101:C=1977
20 PRINT "FIRST TIME";A;B;C
30 CLEAR
40 PRINT "SECOND TIME";A;B;C
50 STOP
```

Here's a PRINTout:

```
FIRST TIME 76 101 1977
SECOND TIME 0 0 0
```

The Statement CLEAR sets the current content of all variables to

zero

17. Throughout our examples in the course so far, we have ended each program with the Statement STOP. There is a reason for having chosen STOP, but you should also learn about another Statement that is used to mark the end of a program: END.

When we first started discussing the BASIC programming language, STOP was easy to understand; everyone knows what it means. END would have been more difficult to comprehend and, actually, the difference between the two Statements is slight. As a matter of fact, it is often not necessary to use either STOP or END since, if they are both omitted, most versions of BASIC will furnish their own END and halt execution of the program when all line numbers have been executed.

STOP or END are needed when it is necessary to mark the **logical end** of a program when it is not the same as the **physical end**.

The physical and logical end of this example program are the same:

```
10 A=76:B=101
20 PRINT A;"TROMBONES AND";B;"CORNETS MAKE LOTS OF NOISE"
```

Most versions of BASIC, including BENTON HARBOR BASIC, do not require a STOP or an END Statement in the example program because the physical and \_\_\_\_\_ ends of the program are the same.

logical



21. After you have written a program and are trying to find out why it doesn't work exactly as you think it should (called "debugging" the program), it's handy to place STOP Statements at one or more significant points. Since the STOP Statement causes BASIC to return to the command level **without resetting** the execution pointer, you can examine or change the contents of any variable and then resume program execution with the CONTINUE Statement.

Here is a demonstration:

```
10 A=76
20 B$=STR$(A)
25 STOP
30 C$=B$+"TROMBONES"
40 PRINT C$
50 END
```

The program RUNs like this:

\*RUN

STOP AT LINE 25

\*PRINT B\$

76

\*B\$="ELEVENTY-TWO "

\*CONTINUE

ELEVENTY-TWO TROMBONES

The Statements STOP and END both terminate execution of a program and cause BASIC to return to the \_\_\_\_\_ level.

command

Turn to your Workbook and perform the exercises labeled "Exercise Group 9". Then return to this page.

This marks the end of our discussion about the “tools” of the BASIC programming language. We have tried to cover the most commonly-used Statements and features, but different versions of BASIC might offer one or more that we haven’t considered. For the most part, these will be unique for the operation of a particular brand of computer. But once you have grasped the general concept of programming Statements presented here, it will be easy to understand the use of any “specials” that your BASIC user’s manual may describe.

For that matter, even though our subject version of BASIC has been the BENTON HARBOR BASIC implementation, there are a couple of “specials” in it that we haven’t covered. Read the user’s manual!

Conversion of programs written for one version of BASIC so that they will RUN on another version involves two things:

1. Mark any Statements whose syntax is different from the version you are using and change them. For example, one BASIC uses INP(X) for obtaining data directly from one of the computer’s “ports” while BENTON HARBOR BASIC uses PIN(X).
2. Mark any Statements that are not implemented in your version of BASIC and find a way to do the same thing with the Statements you do have. For example, one BASIC has the Statement MOD for doing modulus arithmetic. It works this way:

`X MOD Y`

returns the remainder after X is divided by Y. But you can easily DEFINE a Function to do the same thing:

```
DEF FN M(X,Y)=X-(Y*INT(X/Y))
```

As another example, a few BASIC’s include the Function FPT(X), which returns the decimal portion of the argument X. This value can easily be extracted by the Statement

```
Y=X-INT(X)
```

We have summarized the Statements, ASCII Number Codes, Relational Operators, Operator Precedence, and Direct Command Statements for you on heavyweight Reference Cards. These cards will quickly show you the proper syntax and provide examples of use while you are creating programs. Keep them nearby while you work at the computer. For more detailed information on any topic, refer to the proper pages in this book (refer to the Index on Page 18-1).

Thank you for your attention. After you have taken the "Review Test", it's on to creativity. Part 2 addresses itself to how your programming tools can be put to good use.



## REVIEW TEST

1. When you are assigning numeric values or strings to variables, the use of the keyword LET is \_\_\_\_\_.
2. If you omit the keyword LET when assigning a numeric value or string to a variable, you must maintain the proper \_\_\_\_\_ for an assignment Statement.
3. In constructing a Relational Test Statement, you may omit the keyword THEN if the conditional Statement is \_\_\_\_\_. (A conditional Statement is one which would normally be placed after the keyword THEN.)
4. More than one Statement may be typed on the same program line. (True) (False)
5. When you type more than one Statement on the same program line, you must end each Statement (except the last one) with a \_\_\_\_\_.
6. When a program line contains more than one Statement, only the \_\_\_\_\_ Statement may be used as a "target" for GOTO and GOSUB Statements.
7. When a Statement follows and is on the same program line as a Relational Test, the Statement will not be executed if the Relational Test \_\_\_\_\_.
8. If a Relational Test comes up false, the rest of the program line which follows the Test will be \_\_\_\_\_.
9. DATA Statements (may) (may not) appear on the same program line as other Statements, providing each Statement (except the last one) is ended with a colon.
10. When BASIC PRINTs numeric values, it \_\_\_\_\_ switches to scientific notation if the value is greater or less than certain preestablished limits.
11. When a numeric value is represented in scientific notation, the letter "E" signifies that the following digits are to be taken as the \_\_\_\_\_ of the value 10.

19. Not necessarily ..... (See Pg. 9-16.)
20. Parentheses ..... (See Pg. 9-17.)
21. Result ..... (See Pg. 9-17.)
22. The current content of all program  
variables is set to zero ..... 16
23. Logical ..... 17
24. CONTINUE ..... 20
25. The one following the line number on  
which the Statement STOP appeared .. 20
26. The lowest line number in the program. 20

## INTRODUCTION TO PART II

### Part II

# Monument Building



## INTRODUCTION TO PART II.

A lot of Statements flowed under your eyes between STOP and END; the first and last ones we discussed in Part I. Do you remember them all? Of course not! No one could.

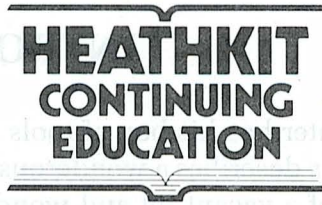
As you put the Statements to use by writing programs, it will get easier for you to pick the right one to accomplish each objective when it comes up. We have compared programming Statements to carpenter's tools. The comparison is particularly apt when you consider how cathedrals are built. Need to drive a nail? Reach for a hammer. Fitting a corner? A mitre box will help.

You may forget about DEF FN X right now, but there will come a time when you need to implement some special mathematical function. Then, as you rummage around your tool box, you'll find good old DEF FN X lying there rusty and unused. Just the one you need!

We truly wish we could assure you that you will become a competent programmer; but neither this nor any other Course can guarantee that. After all, not all carpenters can build cathedrals. What we can do is tackle a couple of projects and hire you on as an apprentice. Not only will you be able to watch and learn, but we'll have you do some of the work on your own. As each project moves on, you will be given more difficult parts to do. Together, we will build something less magnificent than a cathedral, but far more challenging than a doghouse.

A word about how we write a computer program: Our way. You will write programs your way and Sam down the street writes them his way. In short, there are as many ways to write programs as there are people writing programs. If it works, great!

Our way may not be the best way, but the final decision has not been made if there really is a best way. If you come up with a better way to accomplish any of the objectives we will undertake, by all means use it; and take a gold star for perception. If you do, you are on your way to journeyman and later, master of the craft.



# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 10*

### HOW DO I START?

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## INTRODUCTION

A Carpenter has his box of tools and knows how each is used. A set of blueprints describes a wondrous cathedral in intricate detail. He stands in front of a vacant lot and wonders, "Where do I begin?"

An artist has all the brushes and colors he can use. He has learned how to mix the colors and which brush to use for any desired effect. A clear mental image predicts the final masterpiece. He sits in front of a blank canvas and wonders, "Where do I begin?"

A writer knows the precise definition of thousands of words and has a dictionary describing many more. In his mind is the outline of a great novel. He sits in front of a blank sheet of paper and wonders, "Where do I begin?"

A computer programmer understands what each Statement accomplishes and the proper syntax in which each is expressed. A Program Objective describes the desired result in intimate detail. He sits in front of a blank CRT screen and wonders, "Where do I begin?"

It seems that the hardest part of creative endeavor is just a matter of getting started!

The BASIC programming Statements have been explained — now it's time to consider how to approach the creation of a program. We will provide a Program Objective and a blank CRT screen and see if we can help you decide where to begin; and end.



## WHAT SHALL WE DO FIRST?

We saw a slick little pocket calculator the other day. It was capable of little more than the four basic math functions of add, subtract, multiply, and divide. What made it unique, however, was the ability to perform these functions in any of three number bases: hexadecimal (hex), octal, or decimal (dec).

How useful such a gadget would be for someone involved in computers! No more laborious conversion of hexadecimal notation to straight binary and then back to octal with pencil and paper. At the push of a button, 9173 (decimal) bytes of free memory could be converted to the address 23D5 (hexadecimal) or 21725 (octal).

Just before parting with fifty hard-to-come-by dollars for the wondrous toy, we were struck by the thought that BASIC might be put to work to emulate the little calculator. Since we already have an H8 computer and BASIC, we could keep the fifty bucks if the program was successful. (This is called the "Look wife! The computer saves money!" syndrome.) Therefore, to begin this section on creativity, let's write a program to do calculations in different number bases.

## LET'S CALCULATE

When writing a computer program, we believe that the hardest part is not **how** to do it. Most often, deciding exactly **what** is to be done takes more time and thought. It is for this reason that we list the three major requirements for writing a computer program as:

1. A precise **definition** of what the program is to accomplish.
2. A detailed, step-by-step **plan** of how the program goals will be achieved.
3. **Knowledge** of how the programming Statements can be used to implement the step-by-step plan.

Let's apply these requirements to writing:

### THE FOUR-BASE CALCULATOR PROGRAM

First, we need a precise definition of what the program is to accomplish.

## What Should It Do?

Suppose you have a particular memory address in mind and want to tell BASIC about it. You know the address as the hex expression "E060"; but since BASIC only knows about decimal values, you will have to do some fancy pencil work to convert E060 to a value the BASIC will understand.

The first thing we want our program to be able to do is accept an expression in one number base and tell us what that value is in a different base. We have to decide how we should express problems so that the program can give us the desired answer. There are quite a number of ways to do it; we'll just pick one and see how it turns out.

First, we will standardize on the following identifiers to communicate our number base selection to BASIC. There's no reason that you could not select other characters if you wished. These are simply our choices.

T = Decimal base

H = Hexadecimal base

Q = Octal base

S = "Split Octal" base

The reason we chose "T" for decimal instead of "D" is because the letter "D" is a legal character in the hexadecimal numbering system, and we can guess that it will make it easier if the number base identifier characters are unique.

The "Split Octal" numbering system is the one used on the Heath H8 Computer, among others. Please be patient; we'll get back to split octal later.

Our first program requirement, that of converting a value expressed in one number base to the same value expressed in a different base, could be typed in as

```
HE060=T
```

To which the program should provide the answer

```
57440
```

The "H" in our expression tells the program that the following characters are in the hex system, and the "T" indicates that we want the answer in decimal. Another thing our program should be able to do is add, subtract, multiply, or divide two expressions in any base. For example, if we type in the expression

```
H1C2C+HC2C1=H
```

we want the answer

```
DEED
```

to be PRINTed.



Suppose we have in mind a beginning memory address expressed in hex as 1C2C. We also have a program that requires 451 (decimal) bytes of memory. We want to know what the hex address of the end of the program will be. Continuing with the expression format we have chosen, we could type

H1C2C+T451=H

and get the answer

1DEF

Looks like it will be quite useful!

Let's write the program so that when a number base is specified, all operations are carried out in that base until a different base is specified. It would be a good idea to have the program stipulate the current base before we type an expression, and also tell the base in which the answer is stated. Accumulating all the ideas so far, a typical program RUN might look like this:

DEC EXPRESSION: H1C2C+T451=H

1DEF HEX

HEX EXPRESSION: 1C2C+1000=

2C2C HEX

HEX EXPRESSION: Q160140+T1000=Q

162110 OCT

OCT EXPRESSION: H

E448 HEX

In the first line, we add 1C2C (hex) and 451 (decimal) and obtained the result expressed in hex as 1DEF.

In the second line, the base remained in Hex, we added 1C2C (hex) and 1000 (hex), and got the hex sum 2C2C.

In the third line the base was switched, we added 160140 (octal) and 1000 (decimal), and called for the result (162110) to be expressed in octal.

In the fourth line we asked for the last result, 162110 (octal), to be converted to Hex and were provided with the answer E448.

## Develop a Plan

Only after deciding **what** to do can you begin thinking about **how** it is to be done. We have come up with a detailed idea of what the calculator program is to do; let's work out a way to do it.

The Flow Chart, which we introduced in Segment 4, helps you organize the program objectives into smaller, individual tasks. You can then implement each task by writing the appropriate Statements. When each task is complete, you can test it and then integrate it with the previously completed portions. The program is thus "built up" piece-by-piece. Let's develop a Flow Chart for our Calculator Program.

The program begins with the PRINTing of an appropriate heading. This step is represented by the symbol numbered "1" on the "Four Base Calculator Flow Chart" (Figure 10-1). At symbol 2, appropriate default values are established for some of the program variables.

Keyboard INPUT of the expression is represented by symbol 3. And at symbol 4, the program will read each character and divide the typed expression into separate parts. The separate parts consist of:

First value

Second value

Math operation

Any specified number base changes

Sub-expressions are converted to decimal values at symbol 5. Symbol 6 represents the step that will perform the specified math operation on the first and second values. Symbol 7 indicates the place in the program where the calculated result is converted into the desired number base. And finally, the calculated result is PRINTed in the desired number base at symbol 8. The program then loops back to symbol 3 for further calculations (if they are necessary).

Our step-by-step plan is complete. We now have:

1. Defined the program goals as a number of smaller tasks to be performed.
2. Developed a Flow Chart to organize the smaller tasks into a logical order.

Let's implement the blocks on our Chart.

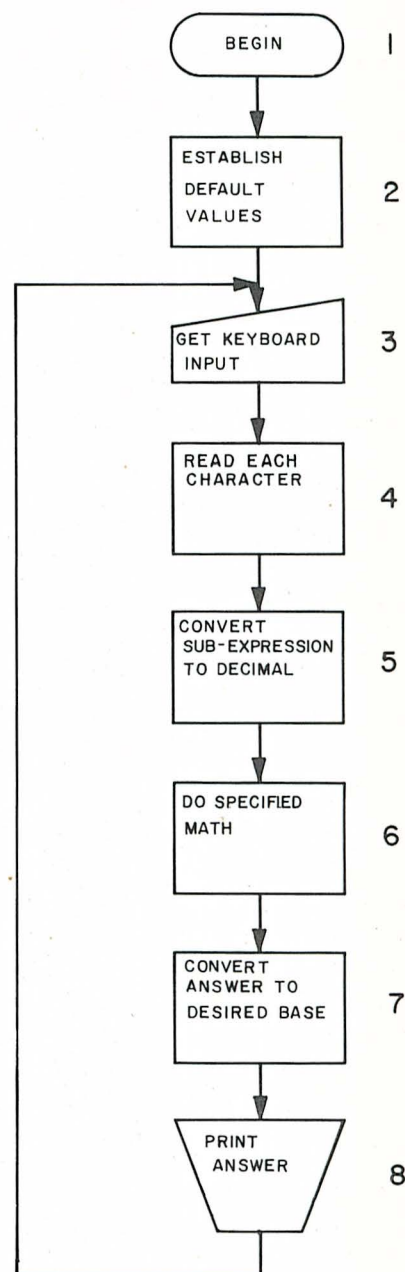


Figure 10-1

The four base calculator flow chart.

## STARTING WITH THE FIRST PART

Our approach to writing programs is to begin with the **most important** task. We call this the “pivotal” part of the program; the one that all other parts depend on or “revolve around”. Most of the time, this pivotal part is obvious. In our calculator program, it is the conversion of a numeric value from one base to another. If this were not the most important part, the program would be trivial since BASIC can easily tell you the sum of 2714 and 3812. All you have to ask is

```
PRINT 2714+3812
```

and BASIC will answer

```
6526
```

In effect, we want to be able to ask

```
HOW MUCH IS 0A9A PLUS 0EE4
```

if hexadecimal notation is being used, or

```
HOW MUCH IS 5232 PLUS 7344
```

if we are working in octal notation. Our program should be able to come up with the answer

```
197E
```

in hex or

```
14576
```

in octal.

Let's agree that number conversion is the pivotal part of our program. That gives us the “where” to start. Now let's look into the “how”.

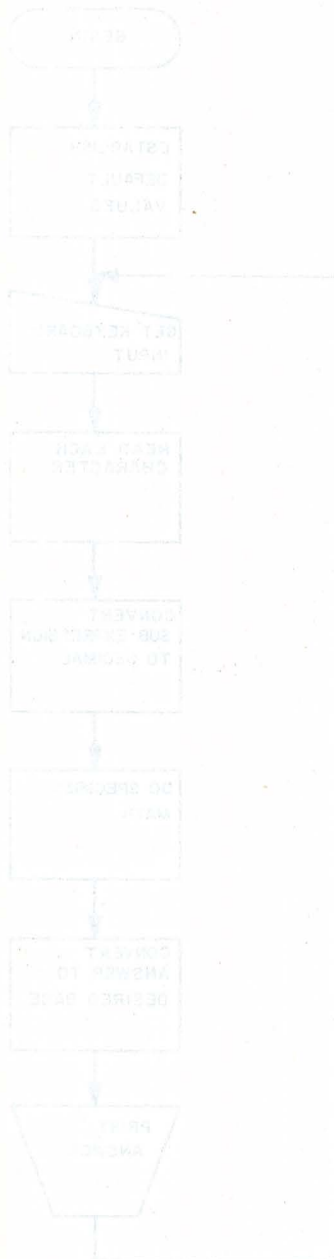


Figure 10-1



## How Shall We Do It?

The instructional Unit that we have included as Appendix B explains in detail the numbering systems that will be used. We will undertake the hexadecimal conversion first.

A value expressed in any number base has two major characteristics:

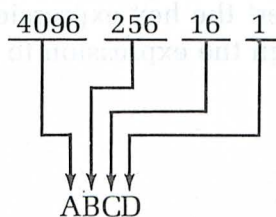
1. A value for each different position in the expression.
2. A value for each different character used in the number base.

The characters used in the hexadecimal number system consist of the familiar numerals 0 through 9 plus the letters A through F. The value of each **character** is as follows:

| HEXADECIMAL |       | HEXADECIMAL |       |
|-------------|-------|-------------|-------|
| Character   | Value | Character   | Value |
| 0           | 0     | 8           | 8     |
| 1           | 1     | 9           | 9     |
| 2           | 2     | A           | 10    |
| 3           | 3     | B           | 11    |
| 4           | 4     | C           | 12    |
| 5           | 5     | D           | 13    |
| 6           | 6     | E           | 14    |
| 7           | 7     | F           | 15    |

Each **position** in a hex expression has a value. In a four-position expression, these values are:

### VALUE OF POSITION



To convert an expression in any number system to an equivalent decimal sum, the value of each position is multiplied by the value of the character occupying that position. The **sum** of all the multiplications is the final result. Thus, the hex expression "0A9A" is equal to the decimal value:

|      |         |      |              |      |
|------|---------|------|--------------|------|
|      | 0 times | 4096 | $0 * 4096 =$ | 0    |
| plus | A times | 256  | $10 * 256 =$ | 2560 |
| plus | 9 times | 16   | $9 * 16 =$   | 144  |
| plus | A times | 1    | $10 * 1 =$   | 10   |

---

For a total of 2714

"OK", you say! "I know all that, but I still don't know how to begin writing a sub-program to convert a hex expression to a decimal value." So what are some other things we know about the problem?

1. Since letters as well as numerals can both appear in a hex expression, the expression must be in the form of a string!
2. We really can't be sure that the string will have exactly four characters. Therefore, it would be best to process the string backwards. In this way, conversion can begin with the lowest valued position and continue with each higher value until there are no more characters in the string.
3. Positional values begin with 1 on the right and each is 16 times the value of the preceding position.

We also need what we call the "**basic premise**" of how we are going to implement the task. The basic premise needed here is how to determine the multiplier value for each of the 16 hex characters.

In programming, there are usually "brute force" ways of doing something and there are also more "elegant" ways which are shorter, faster, and take up less memory. The elegant way is a challenge to your skill as a programmer. But let's consider the "brute force" way first.

Suppose we set out to convert the hex expression 0A9A to a decimal value. First, we need to assign the expression to a string variable:

```
10 LET H$="0A9A"
```

See? It isn't really so hard to begin! Now we need to establish the value of the first position:

```
20 LET X=1
```

Easy as pie!

Next, let's set up a FOR . . . NEXT loop to run through the string backwards. We don't know how long the string will be each time, but BASIC includes a Statement that will measure it for us. Do you remember what it is; the Function that returns the LENgth of a string?

To set up a FOR . . . NEXT loop to process a string backwards, the initial value of the loop must be the last character in the string. The terminal value must be the first character. To make the loop go backwards, the step value must be negative; that is: -1.

```
30 FOR I=LEN(H$) TO 1 STEP -1
```

Now for the body of the loop. Remember, we'll do it the brute force way first. The objective is to multiply the character value by the position value and keep a total of the multiplications. The function MID\$ will extract one character at a time from the string. If we allow the loop variable to choose which one, we can process them all in order, last to first.

```
40 IF MID$(H$,I,1)="0" THEN T=T+(0*X)
50 IF MID$(H$,I,1)="1" THEN T=T+(1*X)
60 IF MID$(H$,I,1)="2" THEN T=T+(2*X)
70 IF MID$(H$,I,1)="3" THEN T=T+(3*X)
80 IF MID$(H$,I,1)="4" THEN T=T+(4*X)
90 IF MID$(H$,I,1)="5" THEN T=T+(5*X)
100 IF MID$(H$,I,1)="6" THEN T=T+(6*X)
110 IF MID$(H$,I,1)="7" THEN T=T+(7*X)
120 IF MID$(H$,I,1)="8" THEN T=T+(8*X)
130 IF MID$(H$,I,1)="9" THEN T=T+(9*X)
140 IF MID$(H$,I,1)="A" THEN T=T+(10*X)
150 IF MID$(H$,I,1)="B" THEN T=T+(11*X)
160 IF MID$(H$,I,1)="C" THEN T=T+(12*X)
170 IF MID$(H$,I,1)="D" THEN T=T+(13*X)
180 IF MID$(H$,I,1)="E" THEN T=T+(14*X)
190 IF MID$(H$,I,1)="F" THEN T=T+(15*X)
```



When the current position value in variable "X" has been multiplied by the character value and totaled, we need to establish the next position value. Since we noted that each position value is 16 times that of the preceding position, all we have to do is multiply the current value of "X" by 16 and assign the new value to "X".

```
200 LET X=X*16
```

(Remember that the computation to the right of the equal sign is made before the current value of "X" is erased.) Now, all that's left is to cycle through the loop until all the characters in the string have been processed.

```
210 NEXT I
```

Try our conversion program on your computer! Just PRINT T to obtain the converted decimal value of the Hexadecimal expression.

### There Must Be a Better Way!

Willard's Law No. 256 states: "In program writing, curiosity never kills cats!" Aren't you curious about how we can use the ASCII number codes that were Charted on Page 8-9 of Segment 8? Let's take a look at those for the characters used in hexadecimal notation.

| HEX<br>CHARACTER | NUMBER<br>CODE |
|------------------|----------------|
| 0                | 48             |
| 1                | 49             |
| 2                | 50             |
| 3                | 51             |
| 4                | 52             |
| 5                | 53             |
| 6                | 54             |
| 7                | 55             |
| 8                | 56             |
| 9                | 57             |
| A                | 65             |
| B                | 66             |
| C                | 67             |
| D                | 68             |
| E                | 69             |
| F                | 70             |

What are some of the things to be observed about the ASCII number codes for the hexadecimal characters?

1. Each hex character has a higher number code than a hexadecimal character with a lower value.
2. Except for a gap of 7 number codes, all the hex characters have consecutive number codes.
3. For the hex characters 0 through 9, if 48 is subtracted from the ASCII number code, what's left is the value of the hex character itself.
4. If more than 9 is left after you subtract 48 from the number code, then subtracting an additional 7 will cause hex characters A through F to have the values of those characters also.

Let's see how that works:

| HEX<br>CHARACTER | NUMBER<br>CODE | MINUS<br>48 | MINUS<br>7 MORE |
|------------------|----------------|-------------|-----------------|
| 0                | 48             | 0           |                 |
| 1                | 49             | 1           |                 |
| 2                | 50             | 2           |                 |
| 3                | 51             | 3           |                 |
| 4                | 52             | 4           |                 |
| 5                | 53             | 5           |                 |
| 6                | 54             | 6           |                 |
| 7                | 55             | 7           |                 |
| 8                | 56             | 8           |                 |
| 9                | 57             | 9           |                 |
| A                | 65             | 17          | 10              |
| B                | 66             | 18          | 11              |
| C                | 67             | 19          | 12              |
| D                | 68             | 20          | 13              |
| E                | 69             | 21          | 14              |
| F                | 70             | 22          | 15              |

Can we use this information somehow? You bet! It leads to the elegant implementation of our conversion sub-program that we talked about earlier.

How shall we write an expression to convert the ASCII number code for a hex character to the value of that character? The answer is given away by the things we know about the number codes listed above. But first we have to convert the hex character to the ASCII number code so we can do some math operations on it. That's because "D" — 48 — 7 isn't allowed in BASIC. But the following is:

```
40 H=ASC(MID$(H$,I,1))-48:IF H>9 THEN H=H-7
```

You may remember from Segment 8 that the function ASC returns the number code of a string character. We use the MID\$ function, just as we did in the brute force example, to select the character from the hex string. Then we subtract 48 from the number code. Next, we test to see if what's left is higher than 9 and, if it is, we subtract 7 more.

When line 40 has been executed, variable 'H' contains the value of the hex character. All that's left is to multiply the position value in variable 'X' by our computed character value and add it to the total:

```
50 T=T+(H*X)
```

Add in the Statement that increments the position value:

```
60 LET X=X*16
```

And finally, the Statement that cycles the loop:

```
70 NEXT I
```

It happens often: When you have written some program Statements to accomplish a task, you will find a shorter, faster, and more elegant way to do it if you keep digging. We can re-write our hex-to-decimal conversion as a sub-program that will become part of the overall project if we add just a couple of features. First, we need to make sure that the "total" variable, 'T', starts out containing zero when we begin the conversion process.

Second, we should check to see that each character in the string is a valid hex character. If a wrong one slips by, the converted value will be off. If we do detect a bad one, let's just set a "flag" so the error can be processed in the main program and thus serve for other conversion sub-programs that we will add.

A "flag" is simply setting a variable to a known value so that it can be tested later. Let's say that variable 'E' is our error variable. If 'E' contains zero, then no error was detected in the conversion process. However, if an error was found, then we will set variable 'E' to 1 and make a premature exit of the conversion loop. Back in the main program, we will test variable 'E' to see if an error was discovered.



Here's the hexadecimal-to-decimal conversion re-written as a sub-program, beginning at line 1000 and including the test for errors:

```
965 REM ** HEXADECIMAL TO DECIMAL CONVERSION SUB-PROGRAM **
975 REM
985 REM HEXADECIMAL EXPRESSION ARRIVES AS CONTENT OF H$
995 REM CLEAR TOTAL & ERROR VARS. SET STARTING POSITION VALUE
1000 T=0:E=0:X=1
1005 REM SET UP LOOP TO RUN BACKWARDS THROUGH STRING
1010 FOR I=LEN(H$) TO 1 STEP -1
1015 REM CALCULATE CHARACTER VALUE
1020 H=ASC(MID$(H$,I,1))-48:IF H>9 THEN H=H-7
1025 REM TEST FOR VALID - SET ERROR FLAG AND EXIT IF NOT OK
1030 IF (H<0 OR H>15) THEN E=1:RETURN
1035 REM MULTIPLY POSITION VALUE BY CHARACTER VALUE AND TOTAL
1040 T=T+(H*X)
1045 REM ADJUST POSITION VALUE
1050 X=X*16
1055 REM LOOP FOR NEXT CHARACTER
1060 NEXT I
1065 REM BACK TO MAIN PROGRAM
1070 RETURN
```

If you type the program into your computer, you may omit the REMark lines to save typing time and memory space. REMark line numbers end in 5 so they are easier to spot. You can also use the BUILD command in BENTON HARBOR BASIC to generate line numbers automatically when typing in our example programs.

One of the best features of BASIC is that you can test the operation of your program as it is being written. For example, you can test our conversion sub-program by adding the following few lines:

```
10 LINE INPUT "HEX EXPRESSION? ";H$
20 GOSUB 1000
30 IF E=1 THEN PRINT "THAT'S A NO-NO!":PRINT:GOTO 10
40 PRINT H$;" IN HEXADECIMAL IS";T;"IN DECIMAL":PRINT:GOTO 10
```

Here's a sample RUN:

```
*RUN
HEX EXPRESSION? 1C2C
1C2C IN HEXADECIMAL IS 7212 IN DECIMAL

HEX EXPRESSION? XXXX
THAT'S A NO-NO!

HEX EXPRESSION? FFFF
FFFF IN HEXADECIMAL IS 65535 IN DECIMAL
```

## How About Vice-Versa?

If we are going to zip around in various number bases, we will also need a conversion sub-program to make hex expressions out of decimal values. It shouldn't be too hard; just the **reverse** of what we have already done. To go from hex to decimal, we multiplied and added to a total. To reverse the procedure, we would subtract from the total, then divide. Since a hex expression containing four characters has a maximum value of 65535 decimal, we will use this as the place to start.

We know that the value of the leftmost position in a four-place hex expression is 4096 decimal. With a little pencil work, we can discover that 65535 divided by 4096 equals 15.999756. What we are really interested in is the 15. The decimal part is what's left over for conversion to Hex characters in other positions.

Let's say that variable "T" has the decimal number that we wish to convert to hex, and variable 'X' has the value of the position in the hex expression that is currently being converted. Line 2000, below, clears out the 'H\$' variable to receive the hex characters as we convert them (H\$="" is for a string variable what H=0 is for a numeric one). Then we set 'X' to the value of the first position we are going to convert. We also clear the error flag and test that the decimal value is in the allowable range. The second line finds out how many "4096's" there are in the decimal number.

```
2000 H$="":X=4096:E=0:IF (T<0 OR T>65535) THEN E=1:RETURN
2010 H=INT(T/X)
```

Reversing what we did in the hex-decimal sub-program, we must now subtract the number of "4096's" from the total so what's left can be converted to hex characters in lower-value positions. Since variable 'H' contains the number of 'X's' in the total, they can be subtracted by adding to line 2010 as follows:

```
2010 H=INT(T/X):T=T-(H*X)
```

Now to convert 'H' to a hex character. Previously, we subtracted 48 from the ASCII number code; this time we'll add 48. Then we tested to see if it was more than 9, and subtracted 7 more if this was the case. This time, we'll test to see if it is more than 57 (the number code for 9) and add 7 if it is.

```
2020 H=H+48:IF H>57 THEN H=H+7
```



Now, convert to an ASCII character and add it to the 'H\$' string.

Remember ol' CHR\$? Bet you thought you'd never use it!

```
2030 H$=H$+CHR$(H)
```

Adjust the positional value in 'X'. In hex-dec we multiplied by 16; guess what's the right thing to do this time.

```
2040 X=X/16
```

Test to see if we are done by adding to line 2040 as follows:

```
2040 X=X/16:IF X<1 THEN RETURN
```

Go for conversion of another position:

```
2050 GOTO 2010
```

Again, we can test what has been done so far. This time, we'll get keyboard entry of a hex expression, convert it to decimal, and save it. Then we can use the dec-hex sub-program to convert back to hexadecimal. If it all works, we should get the same PRINTout as before.

```

5 REM ** TEST PROGRAM FOR HEX-DEC AND DEC-HEX CONVERSION
10 LINE INPUT "HEX EXPRESSION ";H$
20 GOSUB 1000
30 IF E=1 THEN PRINT "THAT'S A NO-NO!":PRINT:GOTO 10
35 REM SAVE CONVERTED DECIMAL VALUE
40 D=T
45 REM GO CONVERT BACK TO HEX
50 GOSUB 2000
55 REM PRINT RESULTS
60 PRINT H$;" IN HEXADECIMAL IS";D;"IN DECIMAL":PRINT:GOTO 10
1000 T=0:E=0:X=1
1010 FOR I=LEN(H$) TO 1 STEP -1
1020 H=ASC(MID$(H$,I,1))-48:IF H>9 THEN H=H-7
1030 IF (H<0 OR H>15) THEN E=1:RETURN
1040 T=T+(H*X)
1050 X=X*16
1060 NEXT I
1070 RETURN
2000 H$="":X=4096:E=0:IF (T<0 OR T>65535) THEN E=1:RETURN
2010 H=INT(T/X):T=T-(H*X)
2020 H=H+48:IF H>57 THEN H=H+7
2030 H$=H$+CHR$(H)
2040 X=X/16:IF X<1 THEN RETURN
2050 GOTO 2010

```



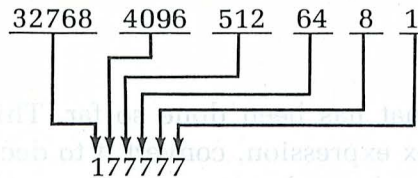
## WHAT'LL WE DO FOR AN ENCORE?

That was fun! While we're on a winning streak, let's see what it takes to convert octal expressions to decimal values.

In the octal number system, we have almost the same conditions as we did in hexadecimal; only the position values and number of characters are different. Octal characters are 0 through 7 and each represents its value. For the maximum decimal number we need to deal with, here are the positional values:

65535 decimal = 177777 Octal

### VALUE OF POSITION



What are some of the things we know about the problem?

1. Even though it consists of decimal-like numbers, an octal expression has a different value than the same sequence of digits taken as decimal. Therefore, the expression must be in the form of a string.
2. We can't be sure how many characters will be in the string; therefore, it must be processed backwards.
3. Positional values begin with 1 on the right, and each is 8 times the value of the preceding position.

Notice that we are considering a "straight" octal expression. The front panel readout on the Heath Model H8 Computer is in what's called "split" octal. Don't worry, we'll get to split octal shortly.

Seeing the similarities between hex and octal expression, we'll bet that you could write an octal-to-decimal conversion sub-program without any further help. Well, maybe just a little help? Fill in each of the following program lines. The correct answers are on Page 10-42.

Clear the total (T) and error (E) variables and set the position value variable (X) to 1.

3000 \_\_\_\_\_

Set up a loop to run through the string backwards. We will stick with 'H\$' for the name of the string variable that contains the octal expression to be converted.

3010 \_\_\_\_\_

Convert the isolated string character to a decimal value using the ASC function minus 48. This time, we won't have to test to see if 7 more need to be subtracted since no letters are used in octal.

3020 \_\_\_\_\_

Test for errors.

3030 \_\_\_\_\_

Add converted character times position value to total.

3040 \_\_\_\_\_

Fix the position value for the next character.

3050 \_\_\_\_\_

And loop until done; then go back home.

3060 \_\_\_\_\_

3070 \_\_\_\_\_

You probably breezed through the last conversion. Keep practicing and you'll be able to tackle the most complex program.

## I'd Rather Do It Myself !

1. OK! Now do one without the suggesting comments. (The correct answers are on Page 10-42.) Beginning with line 4000, write a sub-program to convert the decimal content of variable 'T' to a straight octal expression that is returned as the content of string variable 'H\$'. Use the decimal-hex sub-program for guidance. Remember, all that is really necessary is to reverse all the operations of the octal-decimal conversion procedure.

2. Finally, since all expressions will be in the form of string characters, we even need conversion sub-programs for decimal. First, at line 5000, decimal characters in the 'H\$' string are converted to a value and stored in variable 'T'. Do you remember the proper Statement? Then, at line 6000, a decimal value in variable 'T' is to be converted to string characters which are stored in the 'H\$' string variable. Remember that one?

5000 \_\_\_\_\_

6000 \_\_\_\_\_



## PUT IT ALL TOGETHER

With our conversion sub-programs written and tested, we can wrap it all together now with the control Statements that form the main part of the program. Again, we are faced with the question of where to begin. Let's see if we can define a "basic premise". The expression that will be typed on the keyboard may include one, a few, or all of the following:

One or more identifiers to switch number bases.

One or two separate expressions.

One or two math operators.

Since these items consist of numerals, letters, and symbols, it is necessary for the keyboard INPUT to be received into a string variable. The "most important part" of this section of the program is to "read" the expression typed on the keyboard and process each character in turn. This is the place to begin. Let's take it one step at a time. The correct answers for the questions (Step 1, Step 2, etc.) begin on Page 10-42.

Let's devise an example expression to work on. How about:

H1C2C+T1000=Q

**Step 1** — Assume that the example expression is the current content of string variable 'E\$' (for expression). How would you set up a loop to read each character of the 'E\$' string individually from left to right?

70 \_\_\_\_\_

**Step 2** — What Statement would you write to isolate a single character from the string so it can be tested against a list of things to do? The loop variable should determine which character to isolate so that each can be processed in turn.

Hint: We will have quite a number of tests to make. It will cut down on typing and memory usage to assign the isolated character to a temporary string variable, such as 'X\$'.

80 \_\_\_\_\_

**Step 3** — If the isolated character is one of the number base identifiers which we established earlier 'T', 'H', 'Q', or 'S', we need to store the information as the "current" base. We will use variable 'B' (for base) and select these values of 'B' to indicate the current base:

| IDENTIFIER | BASE      | 'B' VALUE |
|------------|-----------|-----------|
| T          | Decimal   | 1         |
| H          | Hex       | 2         |
| Q          | Octal     | 3         |
| S          | Split Oct | 4         |

What Statements would you write to compare the isolated character against the list of number base identifiers and assign the appropriate value to 'B' if a match is found?

90 \_\_\_\_\_

100 \_\_\_\_\_

110 \_\_\_\_\_

120 \_\_\_\_\_

**Step 4** — If the isolated character was not a number base identifier, it might be a math operator. The next Statements should compare the character in 'X\$' against the list of math operators.

If you find that the isolated character is one of the math operators, an indicator must be stored for later use because of the way the expression is written. For example, if "+" is found, no addition can take place since the second value to be added has not yet been read. We will store an indicator of the desired math operation in variable 'M' as follows. (The math operator "=" is special and will be considered later.)

| MATH OPER | 'M' VALUE |
|-----------|-----------|
| +         | 1         |
| -         | 2         |
| *         | 3         |
| /         | 4         |

What Statements would you write to test for a math operator and assign the proper value to 'M' if one is found?

130 \_\_\_\_\_

140 \_\_\_\_\_

150 \_\_\_\_\_

160 \_\_\_\_\_

**Step 5** — If the character in 'X\$' is not a number base identifier or a math operator, then it must be part of one of the sub-expressions. Later on, we will add a check for illegal characters. For now, we will just consider the very basic things that need to be done.

All our "convert to decimal" sub-programs operate on a string which is received as the current content of variable "H\$". Write the Statement which will assemble a sub-expression by adding characters to the 'H\$' string.

170 \_\_\_\_\_

**Step 6** — Now we need a Statement to cycle through the loop for as many times as there are characters in the 'E\$' string. That would be:

180 \_\_\_\_\_

Here's what we have so far:

```
70 FOR A=1 TO LEN(E$)
80 X$=MID$(E$,A,1)
90 IF X$="T" THEN B=1
100 IF X$="H" THEN B=2
110 IF X$="Q" THEN B=3
120 IF X$="S" THEN B=4
130 IF X$="+" THEN M=1
140 IF X$="-" THEN M=2
150 IF X$="*" THEN M=3
160 IF X$="/" THEN M=4
170 H$=H$+X$
180 NEXT A
```



**Step 7** — Program lines 90 through 120 detect number base changes and store the new base as a discrete value of variable 'B'. If such a change is detected, the identifier shouldn't be added to the sub-expression string that we are assembling in 'H\$'.

What Statement would be proper to add at the end of each of lines 90 through 120 to keep the identifier from being added to the 'H\$' string?

90 \_\_\_\_\_

100 \_\_\_\_\_

110 \_\_\_\_\_

120 \_\_\_\_\_

**Here's our example expression again:**

H1C2C+T1000=Q

So far, our program Statements have taken care of the number base switch to hex as called for by the identifier 'H', and assembled the sub-expression "1C2C" as the content of the 'H\$' string. The next character read is "+". We have already provided a means to detect this character and store an indicator as the value of variable 'M'; in this case, the value is 1. But detecting "+" or any of the other math operators also signals that the sub-expression we have been assembling is complete.

**Step 8** — The base of the sub-expression is described by the value that is the current content of variable 'B'. Since it is possible that the content may be switched for the next sub-expression, we need to spin off and convert this one to decimal before we lose it.

Write a Statement that will convert the string sub-expression that is the current content of variable 'H\$' to decimal by calling on the proper conversion sub-program indicated by the value stored in variable 'B'.

Hint: Root around in Segment 5 and Workbook Experiment Group 8 for some interesting suggestions.

400 \_\_\_\_\_

**Step 9** — We need to add a Statement to each of lines 130 through 160 to jump to the “conversion control” program line (400) just written.

What would that be?

130 \_\_\_\_\_

140 \_\_\_\_\_

150 \_\_\_\_\_

160 \_\_\_\_\_

Now you're really rolling! Keep up the good work.

**Step 10** — After the appropriate conversion sub-program has been executed, as controlled by line 400, the loop should be re-entered for processing of any additional characters that may remain in the expression string.

What's the proper Statement to add to line 400?

400 \_\_\_\_\_

**Step 11** — The math operator “=” requires almost the same program steps as the others, but with one exception: The converted value of the previous sub-expression must be saved so that variable ‘T’ can be available to receive the converted value of the new sub-expression.

Write a program line to save the previous value in variable ‘V’ and jump to the conversion control line (400) when the “=” character is read in the expression string.

162 \_\_\_\_\_

**Step 12** — Everything is complete with the exception of performing the previously stored math operation (content of variable ‘M’) on the values that are the contents of variable ‘V’ (first sub-expression converted to decimal) and variable ‘T’ (second sub-expression converted to decimal).

First, write the Statements that will perform the math operations of add, subtract, multiply, and divide. They should be written as sub-programs.

300 \_\_\_\_\_

310 \_\_\_\_\_

320 \_\_\_\_\_

330 \_\_\_\_\_

**Step 13** — What Statement would you write to select the proper math operation sub-program when all characters of the expression string have been processed?

Hint: Look back to Step 8 for a clue.

190 \_\_\_\_\_

**Step 14** — All that's left is to convert the result of the computation, now the current content of variable 'T', to the last specified number base and PRINT the answer.

What are the proper Statements?

200 \_\_\_\_\_

210 \_\_\_\_\_



Here's the framework of our program as we have developed it so far:

```

70 FOR A=1 TO LEN(E$)
80 X$=MID$(E$,A,1)
90 IF X$="T" THEN B=1:GOTO 180
100 IF X$="H" THEN B=2:GOTO 180
110 IF X$="Q" THEN B=3:GOTO 180
120 IF X$="S" THEN B=4:GOTO 180
130 IF X$="+" THEN M=1:GOTO 400
140 IF X$="-" THEN M=2:GOTO 400
150 IF X$="*" THEN M=3:GOTO 400
160 IF X$="/" THEN M=4:GOTO 400
162 IF X$="=" THEN V=T:GOTO 400
170 H$=H$+X$
180 NEXT A
190 ON M GOSUB 300,310,320,330
200 ON B GOSUB 6000,2000,4000,4000
210 PRINT H$
300 T=V+T:RETURN
310 T=V-T:RETURN
320 T=V*T:RETURN
330 T=V/T:RETURN
400 ON B GOSUB 5000,1000,3000,3000:GOTO 180

```

Finally, before we can test the program, an INPUT Statement is required so we can type in our expression. How about:

```

60 LINE INPUT "EXPRESSION: ";E$

```

## ERRORS PLEASE?

Now we are almost ready to test. But wait! We can spot two errors in the program that will keep it from working. One of them is not too hard to spot; the other is well hidden. The conversion sub-programs are not included in the above listing; but that's not one of the errors we are thinking of.

Can you spot the two errors? Test your programming skill and don't turn to the answer (Page 10-45) until you think you have found them. List what you find below.

Hint: One of the errors was discussed in Segment 9. The other error has not been discussed before. Work hard and find it anyway.

Error #1: \_\_\_\_\_

---

---

This program line will fix the error:

---

Error #2: \_\_\_\_\_

---

---

This program line will fix the error:

---

## TESTING

Type into your computer the program lines listed on Page 10-29, plus the input statement of line 60, the error fixes, and the conversion sub-programs written earlier. After everything is typed, save the program if you are able to do so. We will be adding some formatting and error tests, but the basic program lines written so far will be retained. Test the program by making several RUNs to make sure that everything is working properly.

Here are the results we obtained when we made a few RUNs:

\*RUN

EXPRESSION: H1C2C+T1000=Q  
020024

\*RUN

EXPRESSION: H1C2C+1000=  
2C2C

\*RUN

EXPRESSION: T1000+1000=  
2000

\*RUN

EXPRESSION: T200\*300=  
60000

\*RUN

EXPRESSION: Q177776/2=  
77777



## FANCY FEATURES

When all the basic program features have been implemented with Statements and tested thoroughly, it's time to move on to the final step — adding fancy features, anti-Watsons, and improved formatting.

**Step 15** — You may have uncovered one of the features that has to be added when you made your test RUNs. As the program stands now, the first character of the typed-in expression must be a number base identifier. Can you explain why this is so?

---

---

---

**Step 16** — To solve the problem stated in Step 15, we should establish an initial value for 'B' that will make line 400 work OK. The typed expression may change 'B' to some other value, but it should start out with a legal one. What would you call the initial, legal value of 'B' that may or may not get changed by the program?

---

**Step 17** — As the default number base indicator value of variable 'B', we will pick 1. In this way, the program will always start out in the decimal base. Write a Statement to establish a default number base indicator value for decimal:

40 \_\_\_\_\_

**Chaining** — It would be nice to be able to "chain" operations from one expression to the next. But this "fancy feature" involves a little more thought. That is, if you have typed

H1C2C+1000=

and obtained the result

2C2C

and wish to add 1000 (hex) more without retyping 2C2C, then we can chain a result with further operations by typing

+1000=

for an answer of

3C2C

What's required to accomplish chaining of operations?

1. The final result (in variable 'T') has to be saved before it is destroyed by the final result conversion sub-program.
2. Line 400 should not be executed if there is no expression in 'H\$' to convert.
3. The saved final result should be used as the value of the first sub-expression if no other characters are typed for it.

**Step 18** — Write a Statement to save the final result before it is destroyed by the conversion program. Use 'S' to save the contents of 'T'. Look over the program carefully and decide where to place the new Statement.

**Step 19** — Write a new set of Statements that begin at line 400 to implement requirement 2 mentioned above.

Hint: Don't forget that LEN will measure the length of a string.

400 \_\_\_\_\_

410 \_\_\_\_\_

**Step 20** — Now we need to use the value of the result, which we saved in 'S', as a default value for the first sub-expression. That is, it will be used as the value of the first sub-expression if none is typed.

Again, look over the program and decide where to place your Statement. By the way, now is a good time to change line 220 so that variable 'H\$' is cleared and the program cycles back to the beginning for another round (!).

**Step 21** — Sometimes, a restriction in the way BASIC works can be used to good advantage. Back in Segment 2, we explained that an ON . . . . GOTO Statement would be ignored if the value of the “pointer” variable was zero. The same is true of ON . . . . GOSUB Statements. We can use this fact to good advantage in our calculator program to enable us to change a previous answer to a different base.

We have already added to our program so that the previous result is placed in variable ‘T’ as a default value. Suppose we wanted to type a single character and have that result PRINTed out in a different base. The only thing preventing this is the math operations specified in line 190 which would try to calculate a new result.

But if we establish a default math operator of zero at the beginning of the program cycle, line 190 will be ignored.

Add the appropriate Statement:

50 \_\_\_\_\_

**NOTE:** Now is also a good time to type in the additional program lines that we have written and re-save the program. Then test for proper operation by having the program convert a number of expressions.

**Step 22** — Remember the error “flag” which we set to 1 in our conversion sub-programs if an improper character was found or too large a number was presented for conversion? Before we invite Mr. Watson over to try our calculator, we’d better add some tests for the error flag.

A separate line for the error message is a good idea so that we can jump to it from more than one place in the program. How about:

```
500 PRINT "SORRY, I CAN'T HANDLE ";E$:E=0:GOTO 50
```



Now, where would you put tests for the error flag?

---

---

---

---

**Step 23** — While we're on the subject of anti-Watsons, we had better include a test for legal characters before any are added to the 'H\$' string. What Statements would you add just before line 170 to make sure that only the legal characters '0' through '9' and 'A' through 'F' are assembled in H\$?

---

---

---

**Step 24** — Finally, we can add some formatting to "fancy-up" the PRINTed output of the program. We decided in our objectives that we would have the program stipulate the current number base when asking for an expression to be typed. We'll use these strings:

DEC for decimal

HEX for Hexadecimal

OCT for Octal

OCS for Split Octal

Write the Statements that will store the proper string in variable 'B\$' whenever a number base switch is specified in an expression.

Also, store the proper string in 'B\$' when the default base is established.

40 \_\_\_\_\_

90 \_\_\_\_\_

100 \_\_\_\_\_

110 \_\_\_\_\_

120 \_\_\_\_\_

**Step 25** — Now, modify line 50 to PRINT the name of the current base.

50 \_\_\_\_\_

**Step 26** — To make the final result look better, it should be TABbed over so it is PRINTed beyond the end of the typed expression. About 35 places should do it. Also, add the name of the current number base to the PRINTed result — use a TAB here, too.

210 \_\_\_\_\_

**Is that all?** — Well, it looks like we're done. After adding the fancy features, the anti-Watsons, and the formatting Statements, our tests show that the program meets all the objectives. . . . Do we hear a voice from the rear of the class? We forgot the Split Octal conversion sub-programs? . . . No, we didn't exactly forget them — we were just saving the best part 'til the last.

## TEST YOUR SKILL AT SPLITTING THE OCTALS!

As a Test of your programming skill, modify and/or add to the calculator program so that split octal-to-decimal and decimal-to-split octal conversions will be made when variable 'B' contains 4. We'll give you a little help.

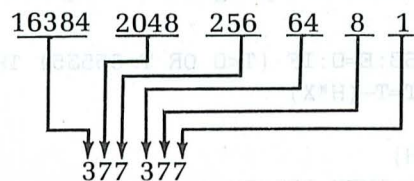
A split Octal numbering system is often used in small computers because it represents a 16-bit address in two number groups. This corresponds to the way a 16-bit address is developed in a small computer; as two 8-bit "bytes" of information.

The Split Octal numbering system simply represents each 8-bit byte separately with a space in between. The largest value that can be represented by 16 binary bits, 65535 (decimal), is written as

377 377

in the split octal system. As with any numbering system, each position has a value. For split octal, these are:

### VALUE OF POSITION



Make a list of what you know about the problem, as we did for the previous problems in this Segment. Think over the items in your list carefully and see if you can develop a "basic premise". Then apply your skill and ingenuity to writing the Statements that will implement the split octal conversions required for the calculator program.

Our list is given on Page 10-47, and our way of working out the Subprogram is given on Page 10-38. (Try to work it out yourself before reading any further.)



## The Split Octal Conversion Sub-Programs

Here's the octal-to-decimal conversion sub-program as previously written:

```

3000 T=0:E=0:X=1
3010 FOR I=LEN(H$) TO 1 STEP -1
3020 H=ASC(MID$(H$,I,1))-48
3030 IF (H<0 OR H>7) THEN E=1:RETURN
3040 T=T+(H*X)
3050 X=X*8
3060 NEXT I
3070 RETURN

```

We reasoned that if the expression being converted were in split octal notation, then when a "space" character appeared, the position value was simply to be set to 256. Processing of the rest of the expression could then continue just as for straight octal. We added this program line to implement our idea:

```

3022 IF H=-16 THEN X=256:GOTO 3060

```

Referring to the ASCII number code Chart, you will see that if you subtract 48 from the number code for a "space", it equals -16. Our new line simply tests for this value.

Here is the decimal-to-octal sub-program as originally written:

```

4000 H$="":X=32768:E=0:IF (T<0 OR T>65535) THEN E=1:RETURN
4010 H=INT(T/X):T=T-(H*X)
4020 H=H+48
4030 H$=H$+CHR$(H)
4040 X=X/8:IF X<1 THEN RETURN
4050 GOTO 4010

```

We can modify this program to serve for split as well as straight octal by adding two lines. First, if variable 'B' contains 4 to indicate that split octal notation is the current base, then the left-most position value should be 16348 instead of the value assigned to 'X' in line 4000. We took care of this by adding the following line:

```
4002 IF B=4 THEN X=16384
```

Then we need to detect when to add a "space" to the expression being assembled as the 'H\$' string. The proper place is when the normal progression of position values reaches 32. At this point, we "fudged" the position value back up to 64 and continued with normal conversion. The following program line does it all:

```
4042 IF B=4 THEN IF X=32 THEN H$=H$+" ":X=64
```

Finally, when the current base is split octal, a space must be allowed as a legal character. Line 162 will reject spaces, but the following will fix it:

```
161 IF B=4 THEN IF X$=" " GOTO 170
```

And the deed is done!

## THE COMPLETE FOUR-BASE CALCULATOR PROGRAM

```
10 PRINT TAB(10); "*** FOUR-BASE CALCULATOR PROGRAM ***"
20 PRINT:PRINT TAB(18); "BY: YOUR NAME"
30 PRINT:PRINT
40 B=1:B$="DEC"
50 M=0:PRINT B$
60 LINE INPUT " EXPRESSION: ";E$
70 FOR A=1 TO LEN(E$)
80 X$=MID$(E$,A,1)
90 IF X$="T" THEN B=1:B$="DEC":GOTO 180
100 IF X$="H" THEN B=2:B$="HEX":GOTO 180
110 IF X$="Q" THEN B=3:B$="OCT":GOTO 180
120 IF X$="S" THEN B=4:B$="OCS":GOTO 180
130 IF X$="+" THEN M=1:GOTO 400
140 IF X$="-" THEN M=2:GOTO 400
150 IF X$="*" THEN M=3:GOTO 400
160 IF X$="/" THEN M=4:GOTO 400
161 IF B=4 THEN IF X$=" " GOTO 170
162 IF X$="=" THEN V=T:GOTO 400
163 IF (X$<"0" OR X$>"F") GOTO 500
164 IF (X$>"9" AND X$<"A") GOTO 500
170 H$=H$+X$
180 NEXT A
190 ON M GOSUB 300,310,320,330
191 IF E=1 GOTO 500
192 S=T
200 ON B GOSUB 6000,2000,4000,4000
201 IF E=1 GOTO 500
210 PRINT TAB(35);H$;TAB(45);B$
220 T=S:H$="":GOTO 50
300 T=V+T:RETURN
310 T=V-T:RETURN
320 T=V*T:RETURN
330 T=V/T:RETURN
400 IF LEN(H$)>0 THEN ON B GOSUB 5000,1000,3000,3000
401 IF E=1 GOTO 500
410 H$="":GOTO 180
500 PRINT "SORRY, I CAN'T HANDLE ";E$:E=0:GOTO 50
1000 T=0:E=0:X=1
1010 FOR I=LEN(H$) TO 1 STEP -1
```



```
1020 H=ASCMID(H$,I,1))-48:IF H>9 THEN H=H-7
1030 IF (H<0 OR H>15) THEN E=1:RETURN
1040 T=T+(H*X)
1050 X=X*16
1060 NEXT I
1070 RETURN
2000 H$="":X=4096:E=0:IF (T<0 OR T>65535) THEN E=1:RETURN
2010 H=INT(T/X):T=T-(H*X)
2020 H=H+48:IF H>57 THEN H=H+7
2030 H$=H$CHR$(H)
2040 X=X/16:IF X<1 THEN RETURN
2050 GOTO 2010
3000 T=0:E=0:X=1
3010 FOR I=LEN(H$) TO 1 STEP -1
3020 H=ASC(MID$(H$,I,1))-48
3022 IF H=-16 THEN X=256:GOTO 3060
3030 IF (H<0 OR H>7) THEN E=1:RETURN
3040 T=T+(H*X)
3050 X=X*8
3060 NEXT I
3070 RETURN
4000 H$="":X=32768:E=0:IF (T<0 OR T>65535) THEN E=1:RETURN
4002 IF B=4 THEN X=16384
4010 H=INT(T/X):T=T-(H*X)
4020 H=H+48
4030 H$=H$+CHR$(H)
4040 X=X/8:IF X<1 THEN RETURN
4042 IF B=4 THEN IF X=32 THEN H$=H$+" ":X=64
4050 GOTO 4010
5000 E=0:T=VAL(H$):RETURN
6000 E=0:H$=STR$(T):RETURN
```

---

We hope you enjoyed writing the Four-Base Calculator Program and urge you to tackle more and more difficult projects to gain the practice that will surely improve your skill and perception. The next Course project should provide just such a challenge.

## ANSWERS

The correct answers are given below for the various questions asked in different parts of this Segment.

### What'll We Do For An Encore

PAGE 10-20

The Octal to Decimal Conversion Sub-program

```

3000 T=0:E=0:X=1
3010 FOR I=LEN(H$) TO 1 STEP -1
3020 H=ASC(MID$(H$,I,1))-48
3030 IF (H<0 OR H>7) THEN E=1:RETURN
3040 T=T+(H*X)
3050 X=X*8
3060 NEXT I
3070 RETURN

```

### I'd Rather Do It Myself

PAGE 10-22

#### 1. The Decimal to Octal Conversion Sub-program

```

4000 H$="":X=32768:E=0:IF (T<0 OR T>65535) THEN E=1:RETURN
4010 H=INT(T/X):T=T-(H*X)
4020 H=H+48
4030 H$=H$+CHR$(H)
4040 X=X/8:IF X<1 THEN RETURN
4050 GOTO 4010

```

#### 2. The decimal to decimal conversion sub-programs

```

5000 E=0:T=VAL(H$):RETURN
6000 E=0:H$=STR$(T):RETURN

```

### Put It All Together

PAGE 10-23

#### STEP 1

```
70 FOR A=1 TO LEN(E$)
```

Using the LEN function as the terminal value of the loop assures us that all characters will be processed, even though we don't know how long a particular expression will be. We used 'A' as a loop variable because 'T' is being used in the conversion sub-programs and we don't want any conflict.

**STEP 2**

```
80 X$=MID$(E$,A,1)
```

The first argument to MID\$ specifies the name of the string from which the character is to be isolated. The second argument selects the position in the string at which isolation is to begin. Using the loop variable for this argument will cycle through each character in turn. The third argument specifies how many characters are to be isolated.

String variable 'X\$' holds the isolated character while we make our tests.

**STEP 3**

```
90 IF X$="T" THEN B=1
100 IF X$="H" THEN B=2
110 IF X$="Q" THEN B=3
120 IF X$="S" THEN B=4
```

Note how much simpler it is to compare 'X\$' to the list instead of repeating the MID\$ Statement each time.

**STEP 4**

```
130 IF X$="+" THEN M=1
140 IF X$="-" THEN M=2
150 IF X$="*" THEN M=3
160 IF X$="/" THEN M=4
```

**STEP 5**

```
170 H$=H$+X$
```

The argument H\$+X\$ adds the new character (in 'X\$') to other characters that have already been assembled into 'H\$' by concatenation.

**STEP 6**

```
180 NEXT A
```

**STEP 7**

```
90 IF X$="T" THEN B=1:GOTO 180
100 IF X$="H" THEN B=2:GOTO 180
110 IF X$="Q" THEN B=3:GOTO 180
120 IF X$="S" THEN B=4:GOTO 180
```



Adding the Statement GOTO 180 at the end of these lines illustrates the fact that program jumps can be made within a loop as we discussed in Segment 6.

**STEP 8**

```
400 ON B GOSUB 5000,1000,3000,3000
```

Remember that variable 'B' contains 1 if the base is decimal. The line number in position 1 of the argument list is that of the decimal string to decimal value conversion sub-program. The line number in position 2 converts hex strings to decimal, and so on.

**STEP 9**

```
130 IF X$="+" THEN M=1:GOTO 400
140 IF X$="-" THEN M=2:GOTO 400
150 IF X$="*" THEN M=3:GOTO 400
160 IF X$="/" THEN M=4:GOTO 400
```

This illustrates the fact that you can make program jumps outside of a loop.

**STEP 10.**

```
400 ON B GOSUB 5000,1000,3000,3000:GOTO 180
```

This illustrates that a loop may be re-entered providing any other loops that were begun after exiting have been completed.

**STEP 11**

```
162 IF X$="=" THEN V=T:GOTO 400
```

**STEP 12**

```
300 T=V+T:RETURN
310 T=V-T:RETURN
320 T=V*T:RETURN
330 T=V/T:RETURN
```

**STEP 13**

```
190 ON M GOSUB 300,310,320,330
```

## STEP 14

```
200 ON B GOSUB 6000,2000,4000,4000
210 PRINT H$
```

## ERRORS Please!

PAGE 10-30

Here are the two errors we were thinking of:

Error #1: After PRINTing the final answer, there is no Statement to mark the **logical end** of the program. In this case, program execution will continue with line 300 and BASIC will encounter a RETURN Statement for which there is no active GOSUB. Remember, GOSUB . . . RETURN must always be used as a Statement pair; you can't have one without the other.

This program line will fix the error:

```
220 STOP
```

Or:

```
220 END
```

Error #2: After the first sub-expression has been converted to decimal, the string still **remains** as the content of variable 'H\$'. Since program line 170 adds the character in 'X\$' to any others that are already in 'H\$', the second sub-expression will be added to the end of the first sub-expression instead of being assembled as a complete, new sub-expression.

This program line will fix the error:

```
400 ON B GOSUB 5000,1000,3000,3000:H$="":GOTO 180
```

## Fancy Features

PAGE 10-32

## Step 15

Because the command RUN sets all variables to zero and the ON . . . GOSUB Statement at line 400 will be ignored when variable 'B' contains the value of zero.

**STEP 16**

A "default" value

**STEP 17**

40 B=1

**STEP 18**

192 S=T

**STEP 19**

400 IF LEN(H\$)>0 THEN ON B GOSUB 5000,1000,3000,3000  
410 H\$="":GOTO 180

**STEP 20**

220 T=S:H\$="":GOTO 40

**STEP 21**

50 M=0

**STEP 22**

191 IF E=1 GOTO 500  
201 IF E=1 GOTO 500  
401 IF E=1 GOTO 500

**STEP 23**

163 IF (X\$<"0" OR X\$>"F") GOTO 500  
164 IF (X\$>"9" AND X\$<"A") GOTO 500

**STEP 24**

40 B=1:B\$="DEC"  
90 IF X\$="T" THEN B=1:B\$="DEC":GOTO 180  
100 IF X\$="H" THEN B=2:B\$="HEX":GOTO 180  
110 IF X\$="Q" THEN B=3:B\$="OCT":GOTO 180  
120 IF X\$="S" THEN B=4:B\$="OCS":GOTO 180



**STEP 25**

```
50 M=0:PRINT B$;
```

**STEP 26**

```
210 PRINT TAB(35);H$;TAB(45);B$
```

## Test Your Skill At Splitting The Octals

**PAGE 10-37**

Here are the things we listed that we knew about the problem:

1. Variable 'B' contains 4 when the Split Octal system is the current base.
2. Beginning on the right, the positional values of a split octal expression are 8 times the value of the previous position until the value reaches 64. Then the value of the next higher position is not  $8 \times 64$ , it is 256.
3. After the modification in positional value progression mentioned in 2, the progression resumes as 8 times the value of the previous position.
4. Straight octal and split octal conversions are done in exactly the same way with the exception of the one change in the progression of positional values.

From this list, we developed our "basic premise" that the straight octal conversion sub-programs could be modified to handle either split or straight octal.



**HEATHKIT  
CONTINUING  
EDUCATION**

# **Individual Learning Program**

## **BASIC PROGRAMMING**

*Segment 11*

### **THE GRAND DESIGN**

**EC-1100**

**HEATH COMPANY**  
BENTON HARBOR, MICHIGAN 49022

**Copyright © 1977**  
Heath Company  
All Rights Reserved  
Printed in the United States of America



## INTRODUCTION

The program we have chosen to write next is the card game Blackjack, or "21" as it is sometimes called. We will begin the program here in Segment 11 and complete it in the following Segments. The program will be written so the computer acts as the Dealer to simulate the game as it is played in Las Vegas casinos.

Our source for the correct rules and options is John Scarne, an authority on games of chance and skill. Our guide will be his book, "Scarne's Encyclopedia of Games", published by Harper and Row, Copyright 1973 by John Scarne Games, Inc. With such an unimpeachable source, we'll have no arguments thank you!

The complete rules of the game are given in "Appendix A." If you are not familiar with the game, read these rules before proceeding. And before you start, remember the three major requirements for writing a computer program:

- A. A precise **definition** of what the program is to accomplish.
- B. A detailed, step-by-step **plan** of how the program goals will be achieved.
- C. **Knowledge** of how the programming Statements can be used to implement the step-by-step plan.

## DEFINING THE PROGRAM

We can precisely define the program as follows:

It is to simulate casino play of the game Blackjack. The computer will act as permanent Dealer and Banker. The following features are to be included:

1. Allow from one to seven Players to participate.
2. Keep track of Player's Bankrolls, wins, losses, and bets.
3. Deal cards realistically and truly at random.
4. Have the Dealer check his hole card if his upcard is a Ten, Jack, Queen, or King.
5. If all Players' hands are either bust or Blackjack, the Dealer does not play his hand.

While we are setting some high objectives, we'll add the following "nice" features:

6. Players are addressed by their name during the course of play.
7. A Player may "sit out" a hand.

Finally, with a bow to showmanship, the PRINTouts of the game's progress are to be "formatted" to enhance the entertainment aspect and make a good visual presentation on the CRT or printer.

To keep the size of the program within practical limits, we won't be including "double down" and Insurance betting, or splitting of pairs. The programming techniques we want to present can be adequately illustrated by the basic game we will create.

That pretty well covers our precise definition of what the program is to accomplish. Now let's work out a plan of how we are going to achieve these goals.

## THE PLAN

Until you have developed your own "style", use the following technique to make the step-by-step plan for your computer program. This will let you to see how the program goals will be achieved.

1. Divide the program into parts, each of which performs a complete portion of the task.
2. Select the most important part; the one that everything else depends on for proper operation. Write the Statements that make this "pivotal" part work and test them thoroughly. Then, pick the next most important part and get it working along with the first part. Proceed to the next most important part, and so on.
3. When writing a series of Statements that perform a simple task, ask yourself if this same task might be also needed in other parts of the program. If so, write it as a sub-program.
4. Finally, add formatting, headings to be PRINTed, instructions, and other "gingerbread".

Let's follow our plan and do the first step.



## DIVIDING THE PROGRAM INTO PARTS

Remember the technique called “**flow-charting**”? We discussed it briefly in Segment 4 as a method of organizing the program objectives into a series of tasks to be done. Our Blackjack game can be divided into smaller parts with a flow-chart, and we can then work on each part in its turn.

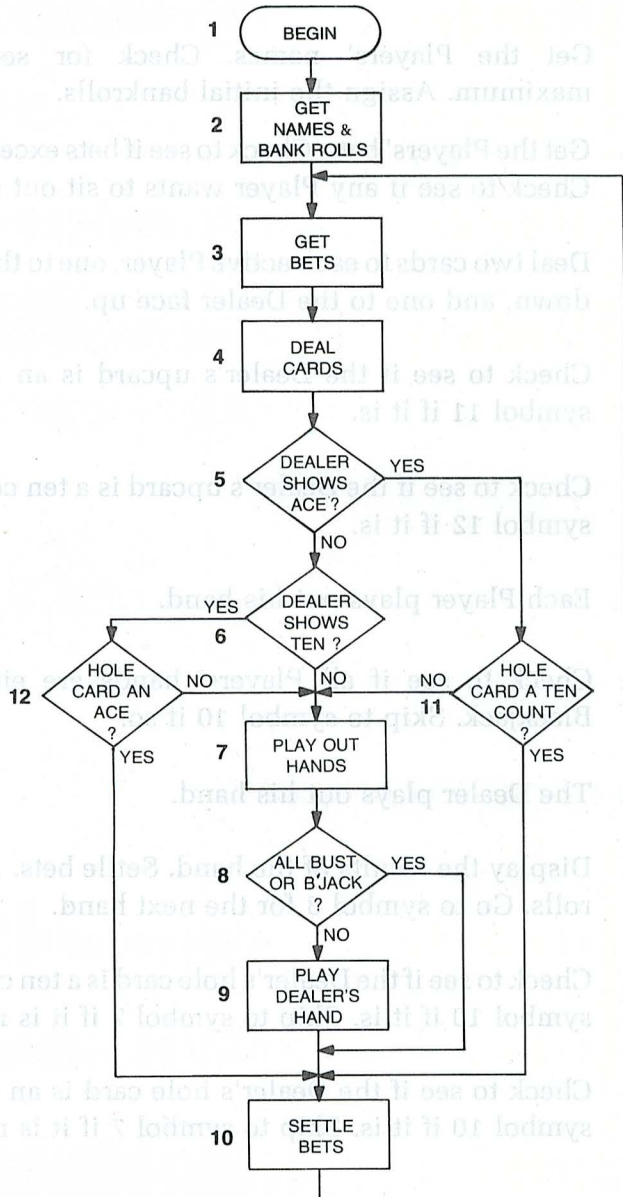


Figure 11-1

A flow-chart of the complete Blackjack game.

Figure 11-1 is an overall flow-chart of the project, showing the progress of the game. Most of the objectives are incorporated in this chart; the remaining ones will be covered within the individual parts. The portions of the program identified by the numbered symbols of the chart are:

- Symbol 1. Begin: PRINT the game heading and rules. DIMension all arrays. Create a deck of cards. Establish strings for card names and suits. Initialize program variables.
- Symbol 2. Get the Players' names. Check for seven Players maximum. Assign the initial bankrolls.
- Symbol 3. Get the Players' bets. Check to see if bets exceed bankrolls; Check to see if any Player wants to sit out this hand.
- Symbol 4. Deal two cards to each active Player, one to the Dealer face down, and one to the Dealer face up.
- Symbol 5. Check to see if the Dealer's upcard is an Ace. Skip to symbol 11 if it is.
- Symbol 6. Check to see if the Dealer's upcard is a ten count. Skip to symbol 12 if it is.
- Symbol 7. Each Player plays out his hand.
- Symbol 8. Check to see if all Players' hands are either bust or Blackjack. Skip to symbol 10 if so.
- Symbol 9. The Dealer plays out his hand.
- Symbol 10. Display the results of the hand. Settle bets. Adjust bankrolls. Go to symbol 3 for the next hand.
- Symbol 11. Check to see if the Dealer's hole card is a ten count. Skip to symbol 10 if it is. Skip to symbol 7 if it is not.
- Symbol 12. Check to see if the Dealer's hole card is an Ace. Skip to symbol 10 if it is. Skip to symbol 7 if it is not.

## DEALING THE CARDS

Now that the program is divided into parts, the next step is to select the most important part. Then we can write the Statements that make it work and test it thoroughly.

Our candidate for the most important part of the Blackjack program is "Deal Cards", symbol 4 in the flow-chart. In fact, it would probably be the starting point for most programs that are based on games using cards. Since we will surely be dealing cards at several different times during a round of play, we will design our card handling system as a sub-program that can be called upon when needed.

### How to Represent the Cards

To construct the card dealing sub-program, we need a basic premise on which to build. Since we have already developed such a premise by trial and error, it would be best to simply explain it.

The first problem we considered was how to represent the cards during the course of play. One of the schemes we have seen used is to display the cards as: AH (for Ace of Hearts), 10S (for Ten of Spades), QD (for Queen of Diamonds), and so on. Our reaction to this type of representation was that it was confusing, not at all a realistic simulation, and that we could do better.

We decided that the cards should be displayed as:

```
ACE OF HEARTS
TEN OF SPADES
QUEEN OF DIAMONDS
```

and so on.

We then considered how all these "strings" for the names and suits of the cards should be stored by BASIC so they could be used when a display of each card was required. The obvious way was a one-dimensional array with each element containing the string for a particular card. Like this:

```
D$(0)="ACE OF DIAMONDS"
D$(1)="TWO OF DIAMONDS"
D$(2)="THREE OF DIAMONDS"
D$(3)="FOUR OF DIAMONDS"
D$(4)="FIVE OF DIAMONDS"
```

and so on for each of the 52 cards in the deck.



Using this technique, each of the 13 card names, such as ACE, would be repeated 4 times, each of the 4 suits, such as DIAMONDS, would be repeated 13 times, and there would be 52 OF's. That's a whole lot of typing, and what's worse, a whole lot of the computer's memory tied up in repetitive string storage. Perceiving this, we set out to find a not-so-obvious way to obtain the name and suit of a card when we wished to display it.

Our basis for card representation depends on **modulus arithmetic**, which is easy to understand. Some versions of BASIC do not include a special function to do this type of arithmetic, but it's easy to DEFine our own once we understand how it works.

When one number is divided by another, there will be a **remainder** ranging from zero to one less than the divisor. For example:

$$36/13=2 \text{ with } 10 \text{ remaining}$$

A MODulo numeric expression returns the remainder only. Thus:

$$36 \text{ MOD } 13=10$$

The expression is read as "Thirty-six MODulo thirteen".

Here are some more examples:

|              |                                                    |
|--------------|----------------------------------------------------|
| 0 MOD 13=0   | (13 "goes into" zero no times with zero remaining) |
| 1 MOD 13=1   | (13 "goes into" one no times with one remaining)   |
| 12 MOD 13=12 | (13 "goes into" 12 no times with 12 remaining)     |
| 13 MOD 13=0  | (13 "goes into" 13 once with zero remaining)       |
| 30 MOD 13=4  | (13 "goes into" 30 twice with 4 remaining)         |

Try these to get the "feel" of modulus arithmetic. The correct answers are on Page 11-23.

|            |       |
|------------|-------|
| 10 MOD 13= | _____ |
| 20 MOD 13= | _____ |
| 30 MOD 13= | _____ |
| 10 MOD 4=  | _____ |
| 20 MOD 4=  | _____ |
| 30 MOD 4=  | _____ |

## Naming the Cards and Suits

The reason we chose modulus arithmetic as our basic premise was prompted by the fact that a deck of cards has:

- 13 different names for cards
- 4 cards of each name
- 4 different suits
- 13 cards of each suit

Seeing the numbers 4 and 13 repeat led us to seek a **numerical relationship**. By experimentation, we found that, if we represented each card by a number in the range 0 through 51, we could apply modulus arithmetic to recover the name and suit of a given card.

Suppose we represent the first few cards of a standard deck by the following numbers:

| <u>NUMBER</u> | <u>CARD</u>      |
|---------------|------------------|
| 0             | ACE OF DIAMONDS  |
| 1             | TWO OF CLUBS     |
| 2             | THREE OF HEARTS  |
| 3             | FOUR OF SPADES   |
| 4             | FIVE OF DIAMONDS |
| 5             | SIX OF CLUBS     |
| 6             | SEVEN OF HEARTS  |
| 7             | EIGHT OF SPADES  |
| 8             | NINE OF DIAMONDS |
| 9             | TEN OF CLUBS     |
| 10            | JACK OF HEARTS   |
| 11            | QUEEN OF SPADES  |
| 12            | KING OF DIAMONDS |
| 13            | ACE OF CLUBS     |
| 14            | TWO OF HEARTS    |
| 15            | THREE OF SPADES  |
| 16            | FOUR OF DIAMONDS |

To see how it's beginning to work, we can state that:

- Suit #0 is DIAMONDS
- Suit #1 is CLUBS
- Suit #2 is HEARTS
- Suit #3 is SPADES

Based on this premise, complete the following list. The correct answers are on Page 11-23.

|          | <u>NUMERIC<br/>VALUE</u> | <u>SUIT NAME<br/>FOR VALUE</u> |
|----------|--------------------------|--------------------------------|
| 4 MOD 4  | 0                        | DIAMONDS                       |
| 12 MOD 4 | 0                        | DIAMONDS                       |
| 11 MOD 4 | _____                    | _____                          |
| 15 MOD 4 | _____                    | _____                          |
| 3 MOD 4  | _____                    | _____                          |
| 5 MOD 4  | _____                    | _____                          |
| 9 MOD 4  | _____                    | _____                          |

By comparing the suit determined by the modulus arithmetic operations to the suits of the cards in the list, we can see that they agree. Thus:

| <u>CARD<br/>NUMBER</u> | <u>MODULO<br/>EXPRESSION</u> | <u>MODULO<br/>VALUE</u> | <u>SUIT NAME<br/>FOR VALUE</u> | <u>SUIT NAME<br/>FROM LIST OF CARDS</u> |
|------------------------|------------------------------|-------------------------|--------------------------------|-----------------------------------------|
| 4                      | 4 MOD 4                      | 0                       | DIAMONDS                       | DIAMONDS                                |
| 12                     | 12 MOD 4                     | 0                       | DIAMONDS                       | DIAMONDS                                |
| 11                     | 11 MOD 4                     | 3                       | SPADES                         | SPADES                                  |
| 15                     | 15 MOD 4                     | 3                       | SPADES                         | SPADES                                  |
| 3                      | 3 MOD 4                      | 3                       | SPADES                         | SPADES                                  |
| 5                      | 5 MOD 4                      | 1                       | CLUBS                          | CLUBS                                   |
| 9                      | 9 MOD 4                      | 1                       | CLUBS                          | CLUBS                                   |

With this proof, we can state that ordering the entire deck in the manner begun above will allow us to extract the suit of the card by modulus arithmetic, where **<card number> MOD 4** indicates the suit when 0=DIAMONDS, 1=CLUBS, 2=HEARTS and 3=SPADES.



## Card Names

Using a similar technique, we can determine the **name** of a card from its card number. Since there are 13 different names, the **<card number> MOD 13** should do it.

If we establish that:

Name # 0 is ACE  
Name # 1 is TWO  
Name # 2 is THREE  
Name # 3 is FOUR  
Name # 4 is FIVE  
Name # 5 is SIX  
Name # 6 is SEVEN  
Name # 7 is EIGHT  
Name # 8 is NINE  
Name # 9 is TEN  
Name #10 is JACK  
Name #11 is QUEEN  
Name #12 is KING

Then:

|           | <u>NUMERIC<br/>VALUE</u> | <u>CARD NAME<br/>FOR VALUE</u> |
|-----------|--------------------------|--------------------------------|
| 4 MOD 13  | 4                        | FIVE                           |
| 12 MOD 13 | 12                       | KING                           |
| 11 MOD 13 | _____                    | _____                          |
| 0 MOD 13  | _____                    | _____                          |
| 13 MOD 13 | _____                    | _____                          |
| 2 MOD 13  | _____                    | _____                          |
| 15 MOD 13 | _____                    | _____                          |

The correct answers are on Page 11-23.

## Card Values

Finally, the value or “**count**” of the card can be determined when needed by the expression  $(\text{<card number> MOD } 13) + 1$ . So that JACKS, QUEENS and KINGS all count 10, we can write:

$$V = (C \text{ MOD } 13) + 1; \text{ IF } V > 10 \text{ THEN } V = 10$$

Where: V is the value of the card  
C is the card number

Write the expressions that will produce the following in accordance with the basic premise we have described. The correct answers are on Page 11-23.

Card Name= \_\_\_\_\_

Card Suit= \_\_\_\_\_

Card Count= \_\_\_\_\_

### ASSIGNMENT 1 — Define a Function

We discussed the way User Functions are DEFINED in Segment 5. Feel free to re-read the information as a memory refresher.

Since our tool box does not include the MOD operator, DEFINE a Function to produce the result:

M1 MOD M2

Name the Function M, for Modulus. A hint is given on Page 11-26 and the correct answer is given on Page 11-24.

## Position Numbers of Cards

In an actual deck, each card may be said to occupy a **position** in the deck. The card at position 12 could be the KING OF DIAMONDS, for example. We could “number” the deck by assigning a value from 0 through 51 to each position. Imagine that the card on the top of the deck is in position 0, the next card is in position 1, the next in position 2, and so on to the last card in position 51.

Keep in mind that the **card number** is completely different from the position number. Thus, while position 12 could be occupied by card number 12, a card with a different number will occupy position 12 as the cards are dealt and moved around.

From what you have learned, what feature of BASIC is suggested by a deck of cards having positions numbered 0 through 51 with a numeric value stored in each position? The correct answer is on Page 11-24.

### ASSIGNMENT 2 — A One-Dimensional Array for 52 Elements

Write the program lines that would create a one-dimensional array named D (for deck) that has 52 elements. Store a different card number in each element of the array. The cards will be numbered from 0 through 51. For now, the card number can be the same as the number of the element in which it is stored. (One-dimensional arrays were discussed in Segment 7.)

It might help if you visualize the array as appearing in memory like this:

| <u>VARIABLE</u> | <u>CONTENTS</u> |
|-----------------|-----------------|
| D(0)            | 0               |
| D(1)            | 1               |
| D(2)            | 2               |
| D(3)            | 3               |
| .               | .               |
| .               | .               |
| .               | .               |
| D(50)           | 50              |
| D(51)           | 51              |

From this point on, we will be writing program lines that will become permanent parts of the final program. You will probably be typing in the program lines as we go along so you can participate in the "Assignments" and "Exercises".

NOTE: We want to include REMark Statements to explain what we have done, but you need not type them as part of the program. To make it easier to identify REMark lines, we will use line numbers ending in 5. That way, you can use the automatic line numbering feature and skip over the REMarks by specifying line number increments of 10.

The correct answer is on Page 11-24

### ASSIGNMENT 3 — Arrays for Names and Suits

Since we are going to use modulus arithmetic to determine what card name and suit are to be PRINTed when we display cards in our game, we need two one-dimensional arrays containing the appropriate strings.



Write the program lines that will store strings for card names and suits in two one-dimensional arrays. Integrate the program lines that set up the card deck in the last Assignment. For element numbers, use the basic premise that we described.

Name the arrays N\$ for names and S\$ for suits.

Here's how you might visualize the arrays:

| <u>VARIABLE</u> | <u>CONTENTS</u> |
|-----------------|-----------------|
|-----------------|-----------------|

|         |            |
|---------|------------|
| N\$(0)  | "ACE"      |
| N\$(1)  | "TWO"      |
| N\$(2)  | "THREE"    |
| N\$(3)  | "FOUR"     |
| N\$(4)  | "FIVE"     |
| N\$(5)  | "SIX"      |
| N\$(6)  | "SEVEN"    |
| N\$(7)  | "EIGHT"    |
| N\$(8)  | "NINE"     |
| N\$(9)  | "TEN"      |
| N\$(10) | "JACK"     |
| N\$(11) | "QUEEN"    |
| N\$(12) | "KING"     |
|         |            |
| S\$(0)  | "DIAMONDS" |
| S\$(1)  | "CLUBS"    |
| S\$(2)  | "HEARTS"   |
| S\$(3)  | "SPADES"   |

A hint is given on Page 11-26 and the correct answer is on Page 11-24.

## EXERCISE A

## Testing What You Have Done So Far

If you have a computer, you can test what you have done so far. This is one of the best features of BASIC; you can write and test small increments of your program before proceeding.

Type in the program lines that you wrote in "Assignment #3." Then, temporarily add lines that will test what you have accomplished so far. (A typical program is shown on the next page.)

The objective PRINTout for your test should resemble this:

\*RUN

CARD NUMBER? 12

THAT CARD IS THE KING OF DIAMONDS

CARD NUMBER? 41

THAT CARD IS THE THREE OF CLUBS

CARD NUMBER? 31

THAT CARD IS THE SIX OF SPADES

CARD NUMBER? 999

END AT LINE 100

\*

## Listing of Our Program for Exercise A

```
10 DIM D(51),N$(12),S$(3)
20 FOR I=0 TO 51:D(I)=I:NEXT I
30 FOR I=0 TO 12:READ N$(I):NEXT I
40 FOR I=0 TO 3:READ S$(I):NEXT I
50 DATA "ACE","TWO","THREE","FOUR","FIVE","SIX","SEVEN"
60 DATA "EIGHT","NINE","TEN","JACK","QUEEN","KING"
70 DATA "DIAMONDS","CLUBS","HEARTS","SPADES"
75 REM DEFINE MODULUS ARITHMETIC FUNCTION
80 DEF FN M(M1,M2)=M1-(M2*INT(M1/M2))
```

```
85 REM ASK FOR CARD NUMBER - PUT IN VARIABLE 'X'
90 INPUT "CARD NUMBER? ";X
95 REM CHECK FOR 'STOPPER' TO END TEST
100 IF X=999 THEN END
105 REM MAKE SURE CARD NUMBER IS IN RANGE
110 IF X<0 OR X>51 GOTO 90
115 REM DO MOD MATH - STORE RESULTS IN 'N' AND 'S'
120 N=FN M(X,13):S=FN M(X,4)
125 REM PRINT NAME AND SUIT OF CARD
130 PRINT "THAT CARD IS THE ";N$(N);" OF ";S$(S)
135 REM LOOP FOR ANOTHER TEST
140 PRINT:GOTO 90
```

All REMark line numbers end in 5 and need not be typed when you are testing. The boxed program lines isolate test Statements from the main program.



## DEALING CARDS (Cont'd)

### Live, Dead, and In-play Cards

At any given time in the card game, we can consider each card as being a member of one of three groups:

1. **Live cards** eligible to be dealt
2. **Dead cards** played in previous hands
3. **In-play cards** currently being played

Visualize the deck as being organized as shown in Figure 11-2. Dividing the deck into these individual groups simplifies keeping track of each card's status.

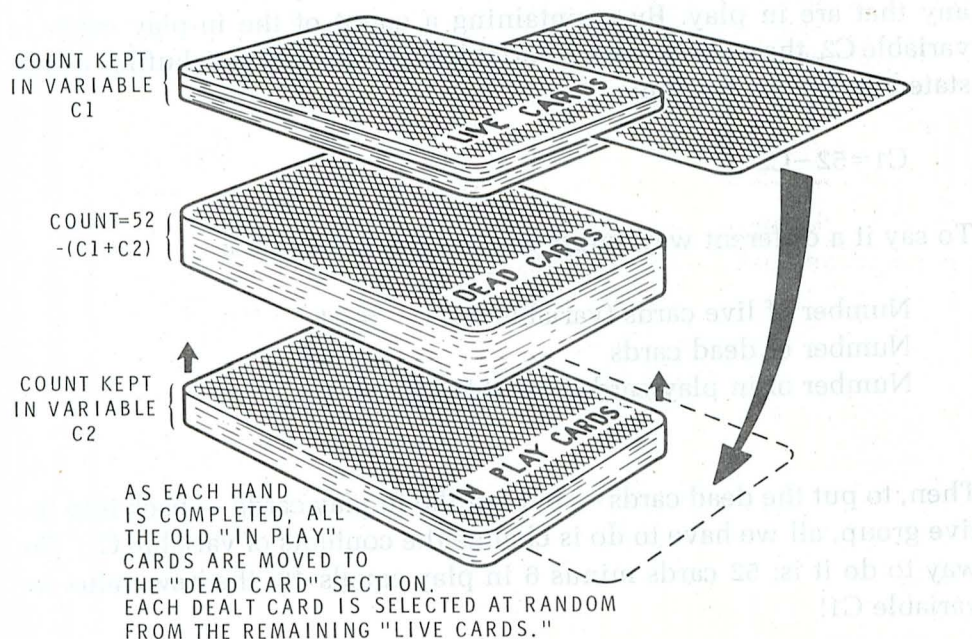


Figure 11-2

To begin the game, you will start with a new deck in which all cards are "live" and eligible to be played. The number of cards in this group will be maintained in variable C1. Initially, the value 52 is assigned to variable C1 when we create the deck of cards.

Add to line 10, as follows, so variable C1 indicates a fresh deck:

```
10 DIM D(51),N$(12),S$(3):C1=52
```

When you wish to deal a card, you must select it from the “live” group. It wouldn’t do to deal a card that has already been played until the deck is shuffled. Nor would it be cricket to deal a card that another Player already held.

Since we will be using the RND function to select cards at random, write a Statement that would restrict the choice to the “live” group only. The correct answer is on Page 11-24. (Check Segment 5 for the way to tell BASIC the limits for the random number you want.)

## Shuffling the Cards

As shown in Figure 11-2, the live group of cards begins at position 0 and extends to the group of dead cards. After the dead cards, there is a group which are in play. When we “shuffle” the deck, we simply change the value of variable C1 so it includes all the previously dead cards — but not any that are in play. By maintaining a count of the in-play cards in variable C2, the value to which variable C1 is to be set to “shuffle” can be stated by the expression:

$$C1 = 52 - C2$$

To say it a different way, suppose the deck is like this:

|                                       |   |    |
|---------------------------------------|---|----|
| Number of live cards (variable C1)    | = | 1  |
| Number of dead cards                  | = | 45 |
| Number of in play cards (variable C2) | = | 6  |

Then, to put the dead cards — but not the in play cards — back into the live group, all we have to do is change the contents of variable C1. The way to do it is: 52 cards minus 6 in play equals 46; the new value for variable C1!

What Statement would you include each time a card is dealt to keep track of the number of cards in play in variable C2? The correct answer is on Page 11-25.

## Dealing Cards

Figure 11-2 also illustrates what happens when a card is dealt from the live group. As it is withdrawn, the position where the card was becomes vacant. Holding the dealt card aside for a moment, all the other cards from that position to the end of the deck are moved up one place to fill in the gap.



Finally, as shown in the Figure, the dealt card is placed at the end of the deck in position 51 where it joins the in-play group.

As each card is dealt, what Statement would you write to keep proper count of the live cards in variable C1? The correct answer is on Page 11-25.

#### ASSIGNMENT 4 — Dealing

Write the program lines that select a card at random from among the live group, move all other cards up to fill in the vacancy, and replace the selected card at the end of the deck.

Here's a visualization that might help you understand what needs to be done:

| Variable | Contents of Variable |            |           |
|----------|----------------------|------------|-----------|
|          | Before Move          | After Move | When Done |
| D(0)     | 0                    | 0          | 0         |
| D(1)     | 1                    | 1          | 1         |
| D(2)     | 2                    | 2          | 2         |
| D(3)     | 3                    | 3          | 3         |
| D(4)     | 4                    | 5          | 5         |
| D(5)     | 5                    | 6          | 6         |
| D(6)     | 6                    | 7          | 7         |
| .        | .                    | .          | .         |
| .        | .                    | .          | .         |
| .        | .                    | .          | .         |
| D(48)    | 48                   | 49         | 49        |
| D(49)    | 49                   | 50         | 50        |
| D(50)    | 50                   | 51         | 51        |
| D(51)    | 51                   | 51         | 4         |

Dealt Card →

If X = 48  
 then:  
 This is D(X) →  
 This is D(X+1) →

Be sure to keep proper count of the number of cards in the live and in-play groups. Since the card-dealing portion of the program will be called upon several times, make it a sub-program with a starting line number of 1000.



## Transferring Cards From In-Play to Dead Group

After each hand has been played, the in-play group of cards can be transferred to the dead group with what Statement?

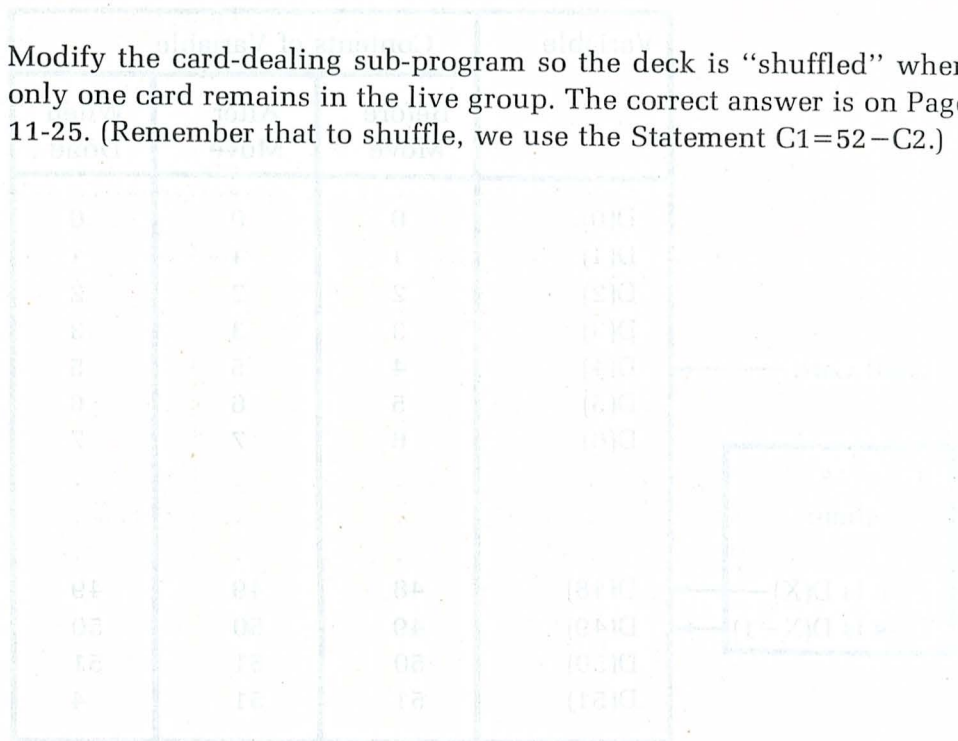
(Remember, variable C2 monitors the number of cards in the in-play group.)

The correct answer is on Page 11-25.

### ASSIGNMENT 5 — Modifying for Shuffle

The program must also have a way of determining when all the live cards in the deck have been dealt and it's time to “**shuffle**”. In a standard game of Black Jack, one card is transferred to the dead group after each shuffle. We will simulate this by shuffling the deck when only one card remains in the live group.

Modify the card-dealing sub-program so the deck is “shuffled” when only one card remains in the live group. The correct answer is on Page 11-25. (Remember that to shuffle, we use the Statement  $C1=52-C2$ .)



## EXERCISE B

### Testing the Card-Dealing Sub-Program

Let's test our card-dealing sub-program and see how it works.

Add some program lines to those we have already written so that 51 cards are dealt and the name and suit of each is PRINTed. Keep a record of each card, as it is dealt, on the tally sheet that we have included. This will verify that the sub-program is working correctly and that each card is dealt once — but only once. When you have dealt 51 cards, you should have a single blank card on your tally sheet.

Including the Statement INPUT X in your test will cause the program to stop and wait for you to type something after it has dealt a card. This gives you time to write down each card after it is dealt. When you're ready, type a number on the keyboard and press RETURN to get the next card.

The number you type isn't used for anything — it's just a way to have the computer to wait for you to get ready.

TALLY SHEET FOR CARD DEALING TEST

| Card Name | DIAMONDS | CLUBS | HEARTS | SPADES |
|-----------|----------|-------|--------|--------|
| ACE       | _____    | _____ | _____  | _____  |
| TWO       | _____    | _____ | _____  | _____  |
| THREE     | _____    | _____ | _____  | _____  |
| FOUR      | _____    | _____ | _____  | _____  |
| FIVE      | _____    | _____ | _____  | _____  |
| SIX       | _____    | _____ | _____  | _____  |
| SEVEN     | _____    | _____ | _____  | _____  |
| EIGHT     | _____    | _____ | _____  | _____  |
| NINE      | _____    | _____ | _____  | _____  |
| TEN       | _____    | _____ | _____  | _____  |
| JACK      | _____    | _____ | _____  | _____  |
| QUEEN     | _____    | _____ | _____  | _____  |
| KING      | _____    | _____ | _____  | _____  |

(A listing of our program is given on the next page.)

## Listing of Our Program for Exercise B

```

10 DIM D(51),N$(12),S$(3):C1=52
20 FOR I=0 TO 51:D(I)=I:NEXT I
30 FOR I=0 TO 12:READ N$(I):NEXT I
40 FOR I=0 TO 3:READ S$(I):NEXT I
50 DATA "ACE","TWO","THREE","FOUR","FIVE","SIX","SEVEN"
60 DATA "EIGHT","NINE","TEN","JACK","QUEEN","KING"
70 DATA "DIAMONDS","CLUBS","HEARTS","SPADES"
80 DEF FN M(M1,M2)=M1-(M2*INT(M1/M2))

```

```

85 REM GO DEAL A CARD
90 GOSUB 1000
95 REM DO MOD MATH TO GET NAME AND SUIT
100 N=FN M(C,13):S=FN M(C,4)
105 REM PRINT NAME AND SUIT OF CARD
110 PRINT N$(N);" OF ";S$(S)
115 REM CHECK IF ALL CARDS DEALT EXCEPT ONE
120 IF C2=51 THEN END
125 REM DUMMY INPUT FOR PROGRAM PAUSE
130 INPUT X
140 GOTO 90

```

```

1000 X=INT(C1*RND(1)):C=D(X)
1010 FOR I=X TO 50:D(I)=D(I+1):NEXT I
1020 D(51)=C:C1=C1-1:C2=C2+1:IF C1=1 THEN C1=52-C2
1030 RETURN

```

Line numbers ending in 5 need not be typed. The boxed program lines isolate test Statements from our main program.

In this Segment, we have **defined** the objectives of the program which is to be written, developed a detailed step-by-step **plan** of how the goals will be achieved, and **created** the first portion of the assignment. In the following Segments we will build on what we have already done by taking each major part of the program and integrating it with the completed parts of the program.



## ANSWERS

The correct answers are given below for the various questions asked in different parts of this Segment.

### Modulus Arithmetic

PAGE 11-8

10 MOD 13=10  
 20 MOD 13=7  
 30 MOD 13=4  
 10 MOD 4=2  
 20 MOD 4=0  
 30 MOD 4=2

### Numeric Value & Suit Names for Values

PAGE 11-9

|          | <u>VALUE</u> | <u>FOR VALUE</u> |
|----------|--------------|------------------|
| 4 MOD 4  | 0            | DIAMONDS         |
| 12 MOD 4 | 0            | DIAMONDS         |
| 11 MOD 4 | 3            | SPADES           |
| 15 MOD 4 | 3            | SPADES           |
| 3 MOD 4  | 3            | SPADES           |
| 5 MOD 4  | 1            | CLUBS            |
| 9 MOD 4  | 1            | CLUBS            |

### Numeric Value & Card Names for Values

PAGE 11-11

|           | <u>NUMERIC<br/>VALUE</u> | <u>CARD NAME<br/>FOR VALUE</u> |
|-----------|--------------------------|--------------------------------|
| 4 MOD 13  | 4                        | FIVE                           |
| 12 MOD 13 | 12                       | KING                           |
| 11 MOD 13 | 11                       | QUEEN                          |
| 0 MOD 13  | 0                        | ACE                            |
| 13 MOD 13 | 0                        | ACE                            |
| 2 MOD 13  | 2                        | THREE                          |
| 15 MOD 13 | 2                        | THREE                          |

### Card Values

PAGE 11-12

Card Name= <card number> MOD 13  
 Card Suit= <card number> MOD 4  
 Card Count=V=(<card number> MOD 13)+1:IF V>10 THEN V=10

**Assignment # 1**

PAGE 11-12

```
DEF FN M(M1,M2)=M1-(M2*INT(M1/M2))
```

**Position Numbers**

PAGE 11-12

A one-dimensional array (or list)

**Assignment # 2**

PAGE 11-13

```
5 REM TELL BASIC THE SIZE OF THE ARRAY
10 DIM D(51)
15 REM USE A LOOP TO ASSIGN VALUES TO ARRAY ELEMENTS
20 FOR I=0 TO 51:D(I)=I:NEXT I
```

**Assignment # 3**

PAGE 11-13

```
5 REM ADD SIZE OF NEW ARRAYS TO DIMENSION STATEMENT
10 DIM D(51),N$(12),S$(3)
15 REM SET UP DECK OF CARDS
20 FOR I=0 TO 51:D(I)=I:NEXT I
25 REM READ NAMES TO N$ LIST - 0 TO 12 IS 13 ELEMENTS
30 FOR I=0 TO 12:READ N$(I):NEXT I
35 REM READ SUITS TO S$ LIST - 0 TO 3 IS 4 ELEMENTS
40 FOR I=0 TO 3:READ S$(I):NEXT I
45 REM HERE ARE THE NAMES AND SUITS
50 DATA "ACE","TWO","THREE","FOUR","FIVE","SIX","SEVEN"
60 DATA "EIGHT","NINE","TEN","JACK","QUEEN","KING"
70 DATA "DIAMONDS","CLUBS","HEARTS","SPADES"
```

(All REMark line numbers end in 5 and need not be typed when you are testing.)

**Live, Dead & In-Play Cards**

PAGE 11-17

```
X=INT(C1*RND(1))
```

## Shuffling the Cards

PAGE 11-18

 $C2=C2+1$ 

## Dealing Cards

PAGE 11-18

 $C1=C1-1$ 

## Assignment # 4

PAGE 11-19

```
995 REM PICK RANDOM NUMBER IN RANGE 0 TO C1
1000 X=INT(C1*RND(1)):C=D(X)
1005 REM START AT THAT POINT AND MOVE ALL CARDS UP ONE PLACE
1010 FOR I=X TO 50:D(I)=D(I+1):NEXT I
1015 REM PUT SAVED CARD AT END - ADJUST VARIABLES
1020 D(51)=C:C1=C1-1:C2=C2+1:RETURN
```

(Sub-program RETURNs with selected card number in variable C.)

## Transferring Cards...

PAGE 11-20

 $C2=0$ 

## Assignment # 5

PAGE 11-20

```
1000 X=INT(C1*RND(1)):C=D(X)
1010 FOR I=X TO 50:D(I)=D(I+1):NEXT I
1015 REM SHUFFLE IF ONLY ONE CARD LEFT IN LIVE GROUP
1020 D(51)=C:C1=C1-1:C2=C2+1:IF C1=1 THEN C*=52-C2
1030 RETURN
```

ADDED STATEMENT





## HINTS

### Defining a Function

PAGE 11-12

The decimal portion of a quotient can be discarded with the Statement:

$$\text{INT}(M1/M2)$$

Example: 26 MOD 7 = 5  
26 divided by 7 = 3.7143  
Integer of 26 divided by 7 = 3  
7 times 3 = 21  
26 minus 21 = 5  
26 MOD 7 = 5

### Assignment # 3

PAGE 11-13

Use the READ . . . . DATA Statement pair. Don't forget to enclose the DATA in quotation marks, since they are strings.

### Assignment # 4

PAGE 11-19

In the "Name the State Capitals" program, we had a requirement to move the elements of an array up. The requirements here are similar.  $D(I)=D(I+1)$  is a good way!



# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 12* **TAKE A CARD**

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## INTRODUCTION

Following our step-by-step plan for writing a computer program, we divided the objective into parts that perform complete portions of the task. Next, we selected what we considered the most important part, wrote the Statements that made it work, and tested it thoroughly.

Now that we have a card-handling system as a base on which to build, it's time to pick the next most important part of the program and make it work.

If we choose the part called "Play Out Hands" (symbol 7 in Figure 12-1, Page 12-3), we will be making decisions that will establish precedents to which the rest of the program must conform. For this reason, "Play Out Hands" seems to be a good choice.



## PLAYING OUT HANDS

After careful consideration of the program objectives and the Rules of Play, we can draw a Flow-Chart (Figure 12-1) that will divide the operation of the "Play Out Hands" function into smaller tasks. This should clarify what we have to accomplish.

The requirements shown by the numbered symbols of the chart are:

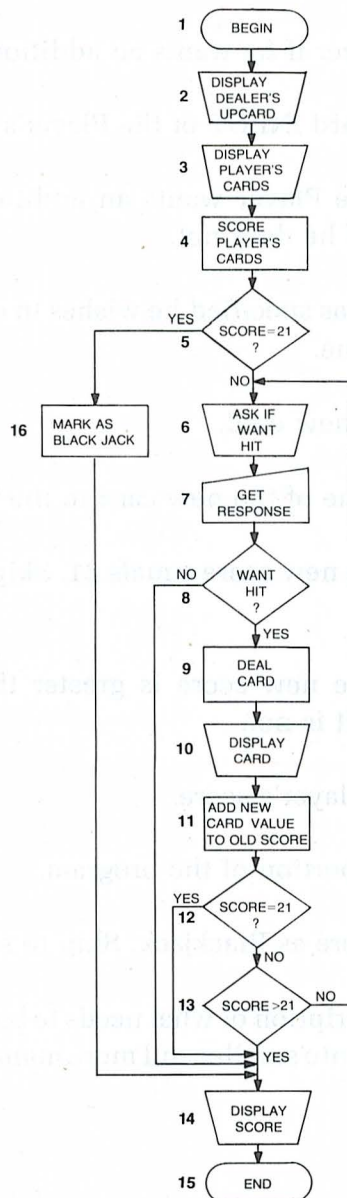


Figure 12-1

The play hands flow-chart.

- Symbol 1. PRINT the Player's name as specified by Objective 6.
- Symbol 2. Display the Dealer's upcard as required by the game rules.
- Symbol 3. Display the two cards dealt to the Player.
- Symbol 4. Calculate the score of the Player's cards.
- Symbol 5. Decide if the Player's score is equal to 21. Skip to symbol 16 if it is.
- Symbol 6. Ask the Player if he wants an additional card.
- Symbol 7. Get a keyboard INPUT of the Player's response.
- Symbol 8. Decide if the Player wants an additional card. Skip to symbol 14 if he does not.
- Symbol 9. The Player has specified he wishes to draw an additional card. Deal one.
- Symbol 10. Display the new card.
- Symbol 11. Add the value of the new card to the Player's score.
- Symbol 12. Decide if the new score equals 21. Skip to symbol 14 if it does.
- Symbol 13. Decide if the new score is greater than 21. Return to symbol 6 if it is not.
- Symbol 14. PRINT the Player's score.
- Symbol 15. End of this portion of the program.
- Symbol 16. Mark the score as Blackjack. Skip to symbol 14.

With a flow-chart and a description of what needs to be done at each step, we have organized our task into smaller and more manageable bites. Let's start at the beginning.

## The Player Information File

Our play hands flow-chart refers to several items we need to know about each Player. These are:

The Player's name

The Player's bankroll

The Player's bet

The two original cards dealt to the Player

The Player's score

To have these items of information available, we need to set up an "Information File" on each Player. If you were to arrange such a file on paper for easy reference, it would probably look like this:

PLAYER INFORMATION FILE

| Player's |         | Orig. Cards |        |      |      |       |
|----------|---------|-------------|--------|------|------|-------|
| No.      | Name    | Bankroll    | Bet    | No.1 | No.2 | Score |
| 0        | WILLARD | 150.00      | 25.00  | 13   | 24   | 21    |
| 1        | LOUIS   | 325.00      | 75.00  | 31   | 44   | 12    |
| 2        | RAINA   | 75.00       | 15.00  | 2    | 37   | 15    |
| 3        | BRUCE   | 225.00      | 30.00  | 17   | 40   | 7     |
| 4        | DAVID   | 94.50       | 24.50  | 26   | 12   | 21    |
| 5        | WILLIAM | 750.00      | 100.00 | 0    | 39   | 12    |
| 6        | RAY     | 200.00      | 20.00  | 4    | 42   | 9     |

One of the most helpful approaches to programming is to find a way to clearly **visualize** in human-oriented form what has to be done, and then translate the visualization into program Statements. The Player Information File is a particularly apt example. Note that Information File has **rows** (one for each Player) and **columns** (one for each item of information to be maintained for each Player). Does the format of this example suggest anything you learned about BASIC? Like a two-dimensional array? Good!



The flow-chart shows that we will need to obtain some information from the Player's file right at the beginning (symbol 3). This is an indication that we had better create the file before we proceed.

We can see that the Player Information File has both **numeric** and **string** entries. Since these two types of data are not compatible in the same array, we will need a one-dimensional array for the names, and a two-dimensional array for the numeric information. Our Program Objectives specify that up to seven Players may participate. Therefore, the name array (we'll call it P\$) needs seven elements and the numeric array (call it P) needs seven rows and five columns.

The portion of the program which we created in the last Segment has some DIMension Statements on line 10. How should it be rewritten to accommodate the arrays we need for our Player Information File?

-----

The correct answer is on Page 12-23.

## The Dealer's Hand

You can also see from the flow-chart that we are going to need some information about the Dealer's hand right away; the second step in this part of the program is to display the Dealer's upcard. On reflection, we only need three items of data about the Dealer:

|                    | <u>VARIABLE<br/>ASSIGNED</u> |
|--------------------|------------------------------|
| Dealer's upcard    | D1                           |
| Dealer's hole card | D2                           |
| Dealer's score     | D3                           |

Simple variables will suffice for these items.

## Player's Names

Now we have a decision to make. In order to test the "play hands" portion of the program as we go along, we have to refer to the Player and Dealer Information Files; but we haven't created them yet. Should we detour and write the portions of the program that set up these files, or should we simulate them and get on with the "play hands" portion?

Because the interaction between different parts of the program sets precedents for the following parts, we vote for the detour. Getting these important files in operation properly will make our tests more meaningful and probably require less modification of program portions as we go along. Let's go to symbol 2 in our overall flow-chart and get the Players' names.

---

### ASSIGNMENT # 1 — Names

Write the portion of the program that obtains the names of the Players. Store the names in a one-dimensional array named P\$. When a Player is named "0", it signifies the end of entry of names. Allow no more than seven Players to participate in the game. (This requires an anti-Watson.)

After you have entered all the names, save the number of active Players in variable N. It should contain zero when there is one Player, 1 when two Players are participating, and so on to a maximum of 6 to indicate seven Players.

Since we have a suspicion that a gathering of names will be required at more than one area of the program, write the Statements as a sub-program. The first line number should be 1040 to conform to the parts of the program we have already written. Also add a Statement at line 90 of the parts of the program we have already written to call on the new sub-program and obtain the Players' names.

The correct answer is on Page 12-23.

NOTE: If your computer is equipped so you can save programs on a storage device such as a cassette recorder, we recommend that you type in the program as written so far and save it. As we progress, we will build on what is already done. You will be able to add each new part as it comes along and test its compatibility with the rest of the program.

## Bankrolls and Bets

### ASSIGNMENT # 2 — Bankrolls

When new Players are brought into the game, as they are when the "Get Names" sub-program is executed, we need to assign them an initial bankroll. Let's keep it a friendly game and make the amount \$150.00.

Remembering that we are going to keep an Information File on each Player as a two-dimensional array named P, we will assign the columns as follows:

Column 0 — Current Bankroll

Column 1 — Bet for this round

Column 2 — Original Card No. 1

Column 3 — Original Card No. 2

Column 4 — Score

Add the necessary Statements to the "Get Names" sub-program to assign each new Player a bankroll of \$150.00. The correct answer is on Page 12-23.

### ASSIGNMENT # 3 — Bets

Now we have the name of each Player, and each has been assigned an initial bankroll. The content of variable N tells us how many Players are participating and will contain zero to indicate one Player, 1 if there are two Players, and so on.

It's time to fill the next slot in each Players' Information File and find out how much they want to bet on this hand. Write the program Statements that will obtain the amount each Player wishes to bet on the current hand. Use a FOR . . . NEXT loop (variable N tells us how many Players there are). Note that if there is only one Player, variable N will contain zero. No problem! A loop that says FOR I=0 TO 0 **will** be executed once.

PRINT the Player's current bankroll before asking for his bet. Include an anti-Watson to make sure the bet is not more than the bankroll. This portion of the program should begin at line 100. The correct answers are on Page 12-23.



## A TEST PRINTOUT

By adding the temporary Statement END at line 170, we can test what is already done before proceeding. Here's the PRINTout we obtained:

\*RUN

PLAYER'S NAME? WILLARD  
PLAYER'S NAME? LOUIS  
PLAYER'S NAME? RAINA  
PLAYER'S NAME? BRUCE  
PLAYER'S NAME? DAVID  
PLAYER'S NAME? WILLIAM  
PLAYER'S NAME? RAY

SEVEN PLAYERS MAXIMUM  
WILLARD

YOUR BANKROLL IS \$150  
YOUR BET? \$200  
BET IS MORE THAN BANKROLL - NO CREDIT!  
YOUR BET? \$25  
LOUIS

YOUR BANKROLL IS \$150  
YOUR BET? \$75  
RAINA

YOUR BANKROLL IS \$150  
YOUR BET? \$15  
BRUCE

YOUR BANKROLL IS \$150  
YOUR BET? \$30  
DAVID

YOUR BANKROLL IS \$150  
YOUR BET? \$24.50  
WILLIAM

YOUR BANKROLL IS \$150  
YOUR BET? \$100  
RAY

YOUR BANKROLL IS \$150  
YOUR BET? \$20

Although the formatting of the PRINTout needs to be improved, the program works properly. We'll leave the formatting until later, as we decided in our step-by-step plan.

## PLAYING OUT HANDS (Cont'd)

### ASSIGNMENT # 4 — Dealing Two Cards

The next items of information we need in our Player's and Dealer's Files are the two cards each is dealt. Since we have already written the card-dealing sub-program, it shouldn't be much of a trick to pass out the cards. Write the program Statements to deal two cards to each Player and two to the Dealer. Use D for a loop variable because the card dealing sub-program uses variable I. From the conventions we established earlier in this Segment, Players' cards go into columns 2 and 3 of the Player Information File. Dealer's cards go into variables D1 and D2.

Do not deal any cards to a Player who has bet zero. It indicates that he wants to sit out the hand, and dealing him cards that won't get played would not be consistent with the realistic simulation we want to maintain.

The correct answer is on Page 12-24.

### ASSIGNMENT # 5 — Playing the Hands

Now that we know all the Player's names, everyone has a bankroll, everyone has placed a bet, and the Dealer and all active Players have been dealt two cards, we can return to the "play hands" part of the program.

Symbol 1 of the "play hands" flow-chart (Figure 12-1) is the beginning of this portion of the program. Since each Player will be involved, a loop is called for. We have also decided that each Player will be referred to by name when it is his turn to play.

Write the program Statements that set up a FOR . . . . NEXT loop so each Player can play out his hand. Use P for the loop variable because the card dealing sub-program uses variable I. We will need to deal cards during the play of hands. Include the Statements that will PRINT each Player's name when it is his turn to play. Begin this portion of the program at line 400 so that there is room for other program portions shown in the overall flow-chart. Remember that variable N holds the number of active Players.

The correct answer is on Page 12-24.

**ASSIGNMENT # 6 — The Dealer's Upcard**

The next symbol in the "play hands" flow-chart calls for PRINTing the Dealer's upcard. Using the technique developed in Exercises 1 and 2 of Segment 11, write a sub-program to PRINT the name and suit of a card. The Statements should be written as a sub-program because a review of the flow-charts drawn so far indicates that we will encounter several occasions where card PRINTing will be required. Pass the number of the card to be PRINTed to the sub-program in variable C. Use line number 1090 to begin the sub-program.

The correct answer is on Page 12-24.

**ASSIGNMENT # 7 — Displaying Names and Suits**

Beginning with line number 420, write Statements that will display the name and suit of the Dealer's upcard and the two cards dealt to the Player. Remember that the Dealer's upcard number is in variable D1 and that the Player's two cards are in columns 2 and 3 of his Information File. For now, format the PRINTing as follows:

DEALER SHOWS FIVE OF DIAMONDS

YOUR CARDS ARE:   TEN OF HEARTS  
                     EIGHT OF CLUBS

↑  
———(PRINT position 20)

The correct answer is on Page 12-24.



## SCORING THE PLAYER'S CARDS

The next symbol in the "play hands" flow-chart, No. 4, calls for the Player's cards to be scored. To implement this portion of the program, we need to detour once again and write the Statements that will calculate scores.

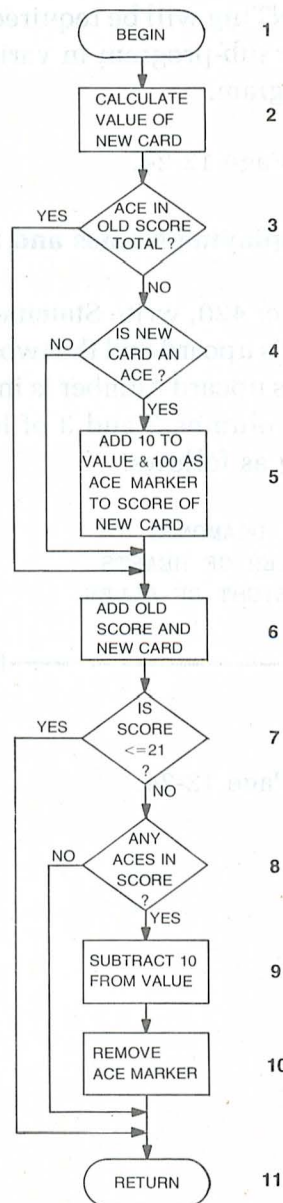


Figure 12-2

A score-calculating flow-chart.

It is a good guess that score calculation will be required at more than one point in the program. Therefore, a sub-program is indicated.

Adding the value of two cards would be easy if it weren't for the fact that an Ace can be counted as either 1 or 11 at the Player's option as described in the game rules. The score-calculating flow-chart (Figure 12-2) will show what has to be done. To make the sub-program useful wherever it might be needed, it will add the value of one card to a previous total passed as the content of variable S.

The requirements shown by the numbered symbols of the chart are:

- Symbol 1. No special requirements to Begin.
- Symbol 2. Using modulus arithmetic as discussed in Segment 11, determine the value of the new card.
- Symbol 3. Determine if the previous score includes an Ace valued at 11. This step is needed because not more than one Ace in a hand can be valued at 11 at the same time. (Two times 11 is 22, which puts the total score over 21.) If the previous score includes an Ace, skip to symbol 6.
- Symbol 4. If the new card is not an Ace, skip to symbol 6.
- Symbol 5. The new card is an Ace. Add 10 to its value, making a total of 11. Also add a "marker" to signify that an Ace is included in the score.
- Symbol 6. Add the previous score and the value of the new card.
- Symbol 7. Is the new total score 21 or less? If so, it's OK; skip to symbol 11 to return to the main program.
- Symbol 8. The score is more than 21. Are there any Aces included? If not, it's the best we can do; skip to symbol 11 to return to the main program.
- Symbol 9. Subtract 10 so all Aces are counted as 1.
- Symbol 10. Delete the marker that indicates an Ace is included in score.
- Symbol 11. Return to the main program.

The score computing flow-chart emphasizes the fact that a complicated problem can be broken down into a series of simple steps. To write the program steps which will compute the score in accordance with the flow-chart, it appears that we need only one new idea; the rest looks pretty straightforward. We have to find a way to **mark** the score as one containing an Ace.

Since modulus arithmetic worked for us before, let's see if it can help us again.

The problem at hand is to have a value represent a score in the range 0 through 31 and, at the same time, carry an indication (or marker) of whether or not an Ace is included. What would happen if we added the value 100 to the score to mark it as one including an Ace? If we did, then:

| <u>SCORE</u> | <u>INDICATION</u>                |
|--------------|----------------------------------|
| 17           | Score is 17 — no Aces included   |
| 117          | Score is 17 — an Ace is included |

Now let's see how we can separate the score from the marker and vice-versa.

Remembering that modulus arithmetic produces the **remainder** after one number is divided by another, we can determine the actual value of the score by the expression:

$\text{<SCORE> MOD } 100$

because:

$117 \text{ MOD } 100 = 17$

Then, to determine if an Ace is included in a score, we can test for our marker by **division**, like this:

When  $\text{INT}(\text{<SCORE>}/100)$  equals 1, an Ace is included



**ASSIGNMENT # 8: — The Score Computing Sub-Program**

Write the score computation sub-program in accordance with the flow-chart. The previous score is passed to the sub-program as the content of variable S. The card to be added to the previous score is passed as the content of variable C. The flow-chart will guide you through each step that needs to be done. Use the premise that the score is marked as one including an Ace by adding 100 to its value.

It's a little complicated, but see if you can do it without any hints. You should be getting better at programming all the time.

Start with line number 1100. The correct answer is on Page 12-25.

**ASSIGNMENT # 9 — Scoring the First Two Cards**

Using the score calculation sub-program, write the Statements that will find the score for the two cards dealt to the Player. Remember that the first card is in column 2 of his Information File, and the second card is in column 3.

The previous total score is passed to the score calculation sub-program as the content of variable S. When the first card is scored, variable S should be set to zero. To score the second card and get the total, the score of the first card (left over in variable S) can be passed back unchanged.

Don't forget, card numbers are passed to the score calculation sub-program as the content of variable C. When the total score is computed, it should be stored in column 4 of the player's information file. Start with line number 450. The correct answer is on Page 12-25.

**ASSIGNMENT # 10 — Check for Blackjack**

After scoring the original two cards dealt to a Player, they should be checked to see if he has "Blackjack" (a score of 21 on the original cards). If the Player has Blackjack, the score should be marked to indicate this fact and distinguish it from a score of 21 obtained by drawing additional cards. The reason for this is because Blackjack wins 1-1/2 times the amount bet.

We need a way to mark a score as being Blackjack! Modulus arithmetic worked well for marking a score as one containing an Ace; why wouldn't it serve as well to mark it as being Blackjack? It's easy to add and easy to test for. Let's add 400 to a score that is Blackjack. Later, when we need to test for it, we can state that if:

$\text{INT}(\text{<SCORE>/400})$  equals 1, it is Blackjack

Now add the Statements to test for Blackjack and mark the score if it is. Use the technique just discussed. If the score is Blackjack, don't forget to store the marked score in the Player's Information File. We will need to know about Blackjack when we settle up the bets.

If the Player has Blackjack, skip to the end of the "play hands" portion of the program where we will PRINT his score. (Since we don't know yet what line number that will be, it will call our attention to the need to fill it in later by using the "\*\*\*" sign to substitute for the line number.)

Line 470 is the next one to use. The correct answer is on Page 12-25.

#### ASSIGNMENT # 11 — Hit Me Dealer !

The next symbol on the "play hands" flow-chart (Figure 12-1) calls for the Player to INPUT a decision to take or not take another card. If he does not want another card, we should GOTO \*\*\* and PRINT his final score. If he does want another card, it should be dealt, PRINTed, and scored. Then we should make the two tests shown in the play hands flow-chart as symbols 12 and 13.

If the Player's score is exactly 21, he has done as well as possible and there's no point in asking if he wants another card. We should GOTO \*\*\* and PRINT his score if this is the case.

If the Player's score is more than 21, he has "gone bust" and his hand is over. If this is the case, we should proceed directly to the PRINT score symbol.

If the Player's score is less than 21, we should loop back to symbol 6 to give him the opportunity to draw an additional card.

Write the program lines that complete the "play hands" portion of the program. Ask if the Player wishes another card. If so, deal a card, score it, and test for exactly 21. If it is 21, go to the score PRINTing portion. If the new card puts his score over 21, the "YOU WENT BUST" message should be PRINTed.

If no additional cards are requested, PRINT the final score.

The last Statement should be the one that will let the next Player play out his hand.

Here's the format (for now) for this assignment:

```
WILLARD
DEALER SHOWS FIVE OF DIAMONDS
YOUR CARDS ARE: FOUR OF CLUBS
 SIX OF HEARTS
HIT? Y
 EIGHT OF SPADES
HIT? N
```

YOUR SCORE IS 18

```
LOUIS
DEALER SHOWS FIVE OF DIAMONDS
YOUR CARDS ARE: TEN OF SPADES
 ACE OF CLUBS
```

YOU HAVE BLACKJACK

```
RAINA
DEALER SHOWS FIVE OF DIAMONDS
YOUR CARDS ARE: KING OF HEARTS
 SIX OF CLUBS
HIT? Y
 NINE OF DIAMONDS
```

YOU WENT BUST

The correct answer is on Page 12-26.



## Zero Bets

One final thing: we forgot to check if the Player had entered zero for a bet, signifying that he wanted to sit out this hand. What Statement would you add to line 400 to make this test and bypass the body of the "play hands" loop when a Player had bet zero?

The correct answer is on Page 12-26.

---

Now's the time to substitute appropriate line numbers for the \*\*\* we used in some GOTO Statements.

```
470 IF S=121 THEN S=421:P(P,4)=S:GOTO 540
```

```
490 IF LEFT$(H$,1)<>"Y" GOTO 540
```

## EXERCISE A

### The Program So Far

Here are all the program lines we have written so far. Add a STOP Statement at line 600 temporarily to mark the logical end of the program, type it into your computer, and test all the features. Be sure to save the program if you have the equipment. We won't have to change much of what we have done, and it's getting big enough that we don't want to keep retyping it.

```
10 DIM D(51),N$(12),S$(3),P$(6),P(6,4):C1=52
20 FOR I=0 TO 51:D(I)=I:NEXT I
30 FOR I=0 TO 12:READ N$(I):NEXT I
40 FOR I=0 TO 3:READ S$(I):NEXT I
50 DATA "ACE","TWO","THREE","FOUR","FIVE","SIX","SEVEN"
60 DATA "EIGHT","NINE","TEN","JACK","QUEEN","KING"
70 DATA "DIAMONDS","CLUBS","HEARTS","SPADES"
80 DEF FN M(M1,M2)=M1-(M2*INT(M1/M2))
90 GOSUB 1040:IF N<0 GOTO 90
100 FOR I=0 TO N
110 PRINT P$(I):PRINT
120 PRINT "YOUR BANKROLL IS $";P(I,0)
130 INPUT "YOUR BET? $":P(I,1)
140 IF P(I,1)<=P(I,0) GOTO 160
150 PRINT "BET IS MORE THAN BANKROLL - NO CREDIT!":GOTO 130
160 NEXT I
170 FOR D=0 TO N
180 IF P(D,1)=0 GOTO 200
190 FOR L=2 TO 3:GOSUB 1000:P(D,L)=C:NEXT L
200 NEXT D
210 GOSUB 1000:D1=C:GOSUB 1000:D2=C
400 FOR P=0 TO N:IF P(P,1)=0 GOTO 590
410 PRINT P$(P)
420 PRINT "DEALER SHOWS ";:C=D1:GOSUB 1090
430 PRINT "YOUR CARDS ARE:":TAB(20);:C=P(P,2):GOSUB 1090
440 C=P(P,3):PRINT TAB(20);:GOSUB 1090
450 S=0:C=P(P,2):GOSUB 1100
460 C=P(P,3):GOSUB 1100:P(P,4)=S
470 IF S=121 THEN S=421:P(P,4)=S:GOTO 540
480 LINE INPUT "HIT? ";H$
490 IF LEFT$(H$,1)<>"Y" GOTO 540
500 GOSUB 1000
510 PRINT TAB(20);:GOSUB 1090
520 GOSUB 1100:P(P,4)=S
530 IF FN M(S,100)<21 GOTO 480
540 PRINT
```

```

550 IF INT(S/400)=1 THEN PRINT "YOU HAVE BLACKJACK":GOTO 580
560 IF FN M(S,100)>21 THEN "YOU WENT BUST":GOTO 580
570 PRINT "YOUR SCORE IS";FN M(S,100)
580 PRINT
590 NEXT P

```

```

600 STOP:REM ** THIS IS TEMPORARY FOR TESTING **

```

```

1000 X=INT(C1*RND(1)):C=D(X)
1010 FOR I=X TO 50:D(I)=D(I+1):NEXT I
1020 D(51)=C:C1=C1-1:C2=C2+1:IF C1=1 THEN C1=52-C2
1030 RETURN
1040 N=-1
1050 LINE INPUT "PLAYER'S NAME? ";P$(N+1)
1060 IF P$(N+1)="0" THEN RETURN
1070 N=N+1:P(N,0)=150:IF N<6 GOTO 1050
1080 PRINT "SEVEN PLAYERS MAXIMUM":RETURN
1090 PRINT N$(FN M(C,13)):" OF ";S$(FN M(C,4)):RETURN
1100 S1=FN M(C,13)+1:IF S1>10 THEN S1=10
1110 IF INT(S/100)=1 GOTO 1140
1120 IF S1<>1 GOTO 1140
1130 S1=S1+110
1140 S=S+S1:IF FN M(S,100)<=21 THEN RETURN
1150 IF INT(S/100)<>1 THEN RETURN
1160 S=S-110:RETURN

```

Here's the PRINTout we obtained when we ran the program:

```

*RUN
PLAYER'S NAME? WILLARD
PLAYER'S NAME? LOUIS
PLAYER'S NAME? RAINA
PLAYER'S NAME? BRUCE
PLAYER'S NAME? DAVID
PLAYER'S NAME? WILLIAM
PLAYER'S NAME? RAY
SEVEN PLAYERS MAXIMUM

```

```

WILLARD
YOUR BANKROLL IS $150
YOUR BET? $25
LOUIS

```

```

YOUR BANKROLL IS $150
YOUR BET? $75
RAINA

```

```

YOUR BANKROLL IS $150
YOUR BET? $15
BRUCE

```



YOUR BANKROLL IS \$150

YOUR BET? \$30

DAVID

YOUR BANKROLL IS \$150

YOUR BET? \$24.50

WILLIAM

YOUR BANKROLL IS \$150

YOUR BET? \$100

RAY

YOUR BANKROLL IS \$150

YOUR BET? \$20

WILLARD

DEALER SHOWS NINE OF CLUBS

YOUR CARDS ARE:      TEN OF HEARTS  
                             EIGHT OF SPADES

HIT? N

YOUR SCORE IS 18

LOUIS

DEALER SHOWS NINE OF CLUBS

YOUR CARDS ARE:      QUEEN OF CLUBS  
                             SEVEN OF CLUBS

HIT? N

YOUR SCORE IS 17

RAINA

DEALER SHOWS NINE OF CLUBS

YOUR CARDS ARE:      EIGHT OF HEARTS  
                             KING OF CLUBS

HIT? N

YOUR SCORE IS 18

BRUCE

DEALER SHOWS NINE OF CLUBS

YOUR CARDS ARE:      FIVE OF CLUBS  
                             TEN OF CLUBS

HIT? Y

JACK OF CLUBS

YOU WENT BUST

DAVID  
DEALER SHOWS NINE OF CLUBS  
YOUR CARDS ARE:      SIX OF DIAMONDS  
                         TWO OF DIAMONDS  
HIT? Y  
                         ACE OF DIAMONDS  
HIT? N  
  
YOUR SCORE IS 19

WILLIAM  
DEALER SHOWS NINE OF CLUBS  
YOUR CARDS ARE:      ACE OF SPADES  
                         KING OF DIAMONDS  
  
YOU HAVE BLACKJACK

RAY  
DEALER SHOWS NINE OF CLUBS  
YOUR CARDS ARE:      JACK OF HEARTS  
                         NINE OF HEARTS  
HIT? N  
  
YOUR SCORE IS 19

STOP AT LINE 600

\*

From the test PRINTout, it appears that the program meets all the objectives for the parts we have written so far. There is still an improvement to be made in the formatting, but that comes later.

In the next Segment, we will add the parts that cause the Dealer to play out his hand. That will give us a little competition and bring the game a step closer to completion.

## ANSWERS

The correct answers are given below for the various questions asked in different parts of this Segment.

### The Player Information File

PAGE 12-6

```
10 DIM D(51),N$(12),S$(3),P$(6),P(6,4):C1=52
```

↑  
added

### Assignment #1

PAGE 12-7

```
85 REM SUB-PROGRAM GETS NAMES - MAKE SURE WE GOT AT LEAST ONE
90 GOSUB 1040:IF N<0 GOTO 90
1035 REM 'N' COUNTS NUMBER OF PLAYERS - START WITH NONE
1040 N=-1
1045 REM ASK FOR NAME
1050 LINE INPUT "PLAYER'S NAME? ";P$(N+1)
1055 REM IF PLAYER NAMED ZERO THEN NO MORE PLAYERS
1060 IF P$(N+1)="0" THEN RETURN
1065 REM COUNT THE NEW PLAYER - CHECK NO MORE THAN 7
1070 N=N+1:IF N<6 GOTO 1050
1075 REM ADVISE NO MORE PLAYERS IF GOT 7
1080 PRINT "SEVEN PLAYERS MAXIMUM":RETURN
```

### Assignment #2

PAGE 12-8

```
1070 N=N+1:P(N,0)=150:IF N<6 GOTO 1050
```

↑  
added

### Assignment # 3

PAGE 12-8

```
95 REM GET BETS FROM ALL PLAYERS - 'N' TELLS HOW MANY
100 FOR I=0 TO N
105 REM LOOP VARIABLE AS SUBSCRIPT PRINTS PROPER NAME
110 PRINT P$(I):PRINT
115 REM LOOP VARIABLE IS PLAYER - ZERO IS BANKROLL COLUMN
120 PRINT "YOUR BANKROLL IS $";P(I,0)
125 REM BET IS STORED IN COLUMN 1 OF INFO FILE
130 INPUT "YOUR BET? $";P(I,1)
135 REM CHECK IF ENOUGH BANKROLL TO COVER BET
140 IF P(I,1)<=P(I,0) GOTO 160
145 REM TELL PLAYER IF NOT ENOUGH BANKROLL
150 PRINT "BET IS MORE THAN BANKROLL - NO CREDIT!":GOTO 130
155 REM LOOP FOR ALL PLAYERS
160 NEXT I
```



## Assignment # 4

PAGE 12-10

Remember that line 170 was added only as a test Statement. The part of the program that we are adding should begin on that line.

```

165 REM SET UP LOOP FOR NUMBER OF PLAYERS
170 FOR D=0 TO N
175 REM CHECK IF HE BET ZERO - COLUMN 1 HAS BET
180 IF P(D,1)=0 GOTO 200
185 REM LOOP DEALS TWO CARDS - SAVE IN COLUMNS 2 AND 3
190 FOR L=2 TO 3:GOSUB 1000:P(D,L)=C:NEXT L
195 REM LOOP FOR ALL PLAYERS
200 NEXT D
205 REM GET TWO CARDS FOR THE DEALER
210 GOSUB 1000:D1=C:GOSUB 1000:D2=C

```

## Assignment # 5

PAGE 12-10

```

400 FOR P=0 TO N
410 PRINT P$(P)

```

## Assignment # 6

PAGE 12-11

```

1085 REM <CARD> MOD 13 GETS NAME SUBSCRIPT - MOD 4 GETS SUIT
1090 PRINT N$(FN M(C,13));" OF ";S$(FN M(C,4)):RETURN

```

Remember that the subscript of a subscripted variable is an expression that can be composed of any mix of real numbers, computations, variable names, or functions. In this case, the subscript can be the User Function which we DEFined to perform modulus arithmetic.

## Assignment # 7

PAGE 12-11

```

415 REM PASS CARD TO SUB-PROGRAM IN VARIABLE C
420 PRINT "DEALER SHOWS ";:C=D1:GOSUB 1090
425 REM PRINT MESSAGE AND PLAYER'S FIRST CARD
430 PRINT "YOUR CARDS ARE: ";TAB(20);:C=P(P,2):GOSUB 1090
435 REM PASS SECOND CARD TO SUB-PROGRAM FOR PRINTING
440 C=P(P,3):PRINT TAB(20);:GOSUB 1090

```

**Assignment # 8****PAGE 12-15**

```
1095 REM VALUE NEW CARD WITH MOD MATH - FACE CARDS COUNT 10
1100 S1=FN M(C,13)+1:IF S1>10 THEN S1=10
1105 REM CHECK PREVIOUS SCORE FOR AN ACE
1110 IF INT(S/100)=1 GOTO 1140
1115 REM CHECK NEW CARD FOR ACE
1120 IF S1<>1 GOTO 1140
1125 REM NEW CARD IS ACE - ADD 10 AND ACE MARKER
1130 S1=S1+110
1135 REM ADD PREVIOUS SCORE AND NEW CARD - RETURN IF 21 OR LESS
1140 S=S+S1:IF FN M(S,100)<=21 THEN RETURN
1145 REM SCORE TOO MUCH - ANY ACES? IF NOT, TOO BAD AND RETURN
1150 IF INT(S/100)<>1 THEN RETURN
1155 REM THERE'S AN ACE - REVALUE AS 1, REMOVE MARKER, RETURN
1160 S=S-110:RETURN
```

**Assignment # 9****PAGE 12-15**

```
445 REM ESTABLISH PREVIOUS SCORE AS ZERO - GET CARD AND SCORE
450 S=0:C=P(P,2):GOSUB 1100
455 REM ADD SCORE OF SECOND CARD AND STORE IN FILE
460 C=P(P,3):GOSUB 1100:P(P,4)=S
```

**Assignment # 10****PAGE 12-15**

```
465 REM CHECK IF BLACKJACK - MARK IF SO AND GO PRINT SCORE
470 IF S=121 THEN S=421:P(P,4)=S:GOTO ***
```

**Assignment # 11****PAGE 12-16**

```
475 REM ASK IF ADDITIONAL CARD IS DESIRED
480 LINE INPUT "HIT? ";H$
485 REM IF NO CARD DESIRED, GO PRINT SCORE AND FINISH
490 IF LEFT$(H$,1)<>"Y" GOTO 540
495 REM ANOTHER CARD WANTED - DEAL IT
500 GOSUB 1000
505 REM PRINT NEW CARD
510 PRINT TAB(20);:GOSUB 1090
515 REM OLD SCORE STILL IN 'S' - ADD NEW CARD - TOTAL TO FILE
520 GOSUB 1100:P(P,4)=S
525 REM IF SCORE IS NOT 21 OR MORE, LOOP BACK
530 IF FN M(S,100)<21 GOTO 480
535 REM PRINT B'JACK OR BUST MESSAGE OR SCORE
540 PRINT
550 IF INT(S/400)=1 THEN PRINT "YOU HAVE BLACKJACK":GOTO 580
560 IF FN M(S,100)>21 THEN PRINT "YOU WENT BUST":GOTO 580
570 PRINT "YOUR SCORE IS";FN M(S,100)
580 PRINT
590 NEXT P
```

**Zero Bets****PAGE 12-18**

```
400 FOR P=0 TO N:IF P(P,1)=0 GOTO 590
```

↑  
— added Statement





# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 13*

### BEAT THE DEALER

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America

## INTRODUCTION

When most people reach this point in the development of a computer program, they begin to feel a sense of excitement. The major hurdles are over; the program is beginning to actually do something; the objectives are being met; and there's only a little more to do before the first three phases of the step-by-step plan are complete.

As the last Exercise in Segment 12 proves, we have developed a Blackjack game that is complete and functional except for these final parts:

Dealer plays out his hand

Bets are settled

When these parts are added, we need only to "dress up" the program by improving the formatting and adding some "gingerbread" features. Let's implement the next part: Dealer plays out his hand. We can't wait to see if our score beats the Dealer's!

## THE DEALER PLAYS HIS HAND

Our program objectives state that the Dealer does not play out his hand if all active Players either have Blackjack or have gone bust.

We have developed a method of marking a Player's score to indicate that it is Blackjack. The "marker" is the value 400 added to the score. In Segment 12, we determined that we could easily test for the Blackjack marker like this:

If  $\text{INT}(<\text{SCORE}>/400)$  equals 1, it is Blackjack

We also developed a method of extracting the absolute score from a value that may contain markers indicating Blackjack or the inclusion of an Ace. The method was:

Absolute score =  $<\text{SCORE}> \text{MOD } 100$

We also DEFINed a Function to perform modulus arithmetic. We pass the score and the modulus value to our user function as arguments to the function call. Here's an example:

$S = \text{FN } M(P(I, 4), 100)$

The Statement will return the absolute value of Player I's score in variable S.

Before we allow the Dealer to play his hand, we must poll each Player's score to determine if it is Blackjack or over 21.

### ASSIGNMENT # 1 — Should the Dealer Play?

Write the program Statements that will poll each active Player's score to determine if it is Blackjack or over 21. Remember to exclude from the poll any Players who have bet zero. A FOR . . . NEXT loop is a good way to do it, with a premature exit to the Dealer's play hand portion when any Player's score is found to be less than 22 and not Blackjack. (Premature exit of a FOR . . . NEXT loop was discussed in Segment 6.)

Since we don't yet know the line number of the Dealer's play hand routine, mark it for attention as "\*\*\*\*". Begin the Statements at line number 600, replacing the temporary STOP Statement we added for testing in Segment 12.

The correct answer is on Page 13-13.



**ASSIGNMENT # 2 — All Blackjacks or Bust**

If we can completely execute the loop written in Assignment #1, it indicates that all Players either have Blackjack or their score is greater than 21 (they went bust). If this is the case, we should PRINT a message stating that the Dealer is not going to play his hand. Program execution should jump directly to the "settle bets" portion if this is the case.

Write the Statements that PRINT an appropriate message if all Player's hands are Blackjack or bust to indicate that the Dealer is not going to play his hand. When the Dealer does not play his hand, GOTO \*\*\* to settle the bets after PRINTing the message. The program lines should begin with number 640. Here's the message format:

```
ALL PLAYERS HAVE BLACKJACK OR WENT BUST
DEALER DOES NOT PLAY HIS HAND
```

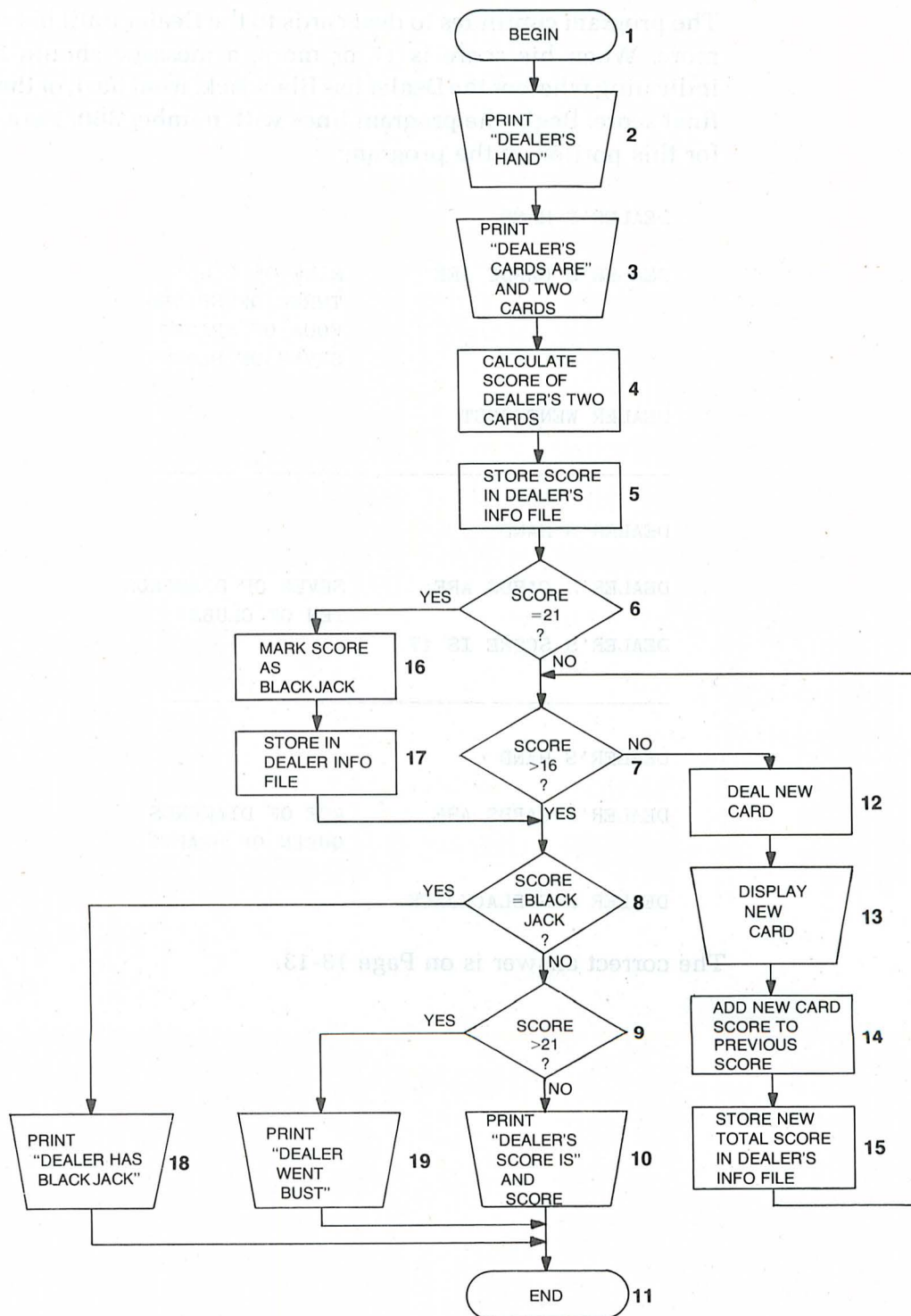
The correct answer is on Page 13-13

**ASSIGNMENT # 3 — The Dealer Plays**

Now it's time to add the Statements that cause the Dealer to play out his hand. According to the rules of the game (Appendix A), the Dealer draws an additional card when his score is 16 or less. If his score after the new card is added is still less than 16, he draws another card. The Dealer continues to draw cards until his score is 17 or higher.

Write the Statements that cause the Dealer to play out his hand. A detailed flow-chart (Figure 13-1) is included with this Assignment as a guide; you're on your own this time.

Call on the sub-program developed in Segment 12 to display the two original cards dealt to the Dealer. When these original two cards have been scored, the score should be tested to determine if it is Blackjack. If it is Blackjack, it should be marked, and the marked score stored in the Dealer's Information File (variable D3). If the score is 16 or less, an additional card should be dealt to the Dealer and added to his score.

**Figure 13-1**

The "Dealer plays his hand" flow-chart.

The program continues to deal cards to the Dealer until his score is 17 or more. When his score is 17 or more, a message should be PRINTed indicating whether the Dealer has Blackjack, went bust, or the value of his final score. Begin the program lines with number 660. Here's the format for this portion of the program:

DEALER'S HAND

DEALER'S CARDS ARE:      NINE OF CLUBS  
                              THREE OF SPADES  
                              FOUR OF SPADES  
                              SEVEN OF HEARTS

DEALER WENT BUST

DEALER'S HAND

DEALER'S CARDS ARE:      SEVEN OF DIAMONDS  
                              TEN OF CLUBS  
DEALER'S SCORE IS 17

DEALER'S HAND

DEALER'S CARDS ARE:      ACE OF DIAMONDS  
                              QUEEN OF HEARTS

DEALER HAS BLACKJACK

The correct answer is on Page 13-13.



## EXERCISE A

### Load All the Program So Far

Since we have added an important part to the program, we should test this new part thoroughly before proceeding. Load the parts of the program we created in Segments 11 and 12 into your computer. Then add all the program lines developed so far in this Segment (except for REMark lines whose numbers end in 5).

Change the GOTO \*\*\* in line 620 to GOTO 660. Change the GOTO \*\*\* in line 650 to GOTO 780.

When the new lines have been typed in, re-save the program on your storage device so you will have an updated copy.

The STOP Statement, temporarily placed at line 780, can be changed if you would like to run through the program a number of times without having to enter Players' names each time. In order to do this, the number of cards in the in play group of the deck must be reset to zero after each hand.

Since variable C2 keeps count of the number of cards that are in play at all times, it should be zero as each new hand is begun. After that is done, you can GOTO 100 to start a new hand, or GOTO 170 to start a new hand without getting bets. Since we are not settling bets as yet, GOTO 170 will allow faster testing.

The temporary Statements for line 780 are:

```
780 C2=0:GOTO 100
```

or, to bypass getting bets:

```
780 C2=0:GOTO 170
```

Here's the PRINTout we obtained in our tests. When you are satisfied that everything is working correctly, continue with this Segment.

\*RUN

PLAYER'S NAME? WILLARD

PLAYER'S NAME? LOUIS

PLAYER'S NAME? DAVID

PLAYER'S NAME? Q

WILLARD

YOUR BANKROLL IS \$150

YOUR BET? \$25

LOUIS

YOUR BANKROLL IS \$150

YOUR BET? \$25

DAVID

YOUR BANKROLL IS \$150

YOUR BET? \$25

WILLARD

DEALER SHOWS THREE OF SPADES

YOUR CARDS ARE: QUEEN OF DIAMONDS  
TEN OF HEARTS

HIT? N

YOUR SCORE IS 20

LOUIS

DEALER SHOWS THREE OF SPADES

YOUR CARDS ARE: QUEEN OF HEARTS  
KING OF CLUBS

HIT? N

YOUR SCORE IS 20

DAVID

DEALER SHOWS THREE OF SPADES

YOUR CARDS ARE: TWO OF HEARTS  
TWO OF CLUBS

HIT? Y

FOUR OF DIAMONDS

HIT? Y

QUEEN OF CLUBS

HIT? N

YOUR SCORE IS 18

DEALER'S HAND

DEALER'S CARDS ARE: THREE OF SPADES  
SEVEN OF SPADES  
EIGHT OF HEARTS

DEALER'S SCORE IS 18

## SETTLING THE BETS

The last major part of the program to be written will decide if the Player has won or lost his bet and adjust his bankroll accordingly. When we have completed this part, the entire program will be working and we can finish with the fancy features and formatting. It looks like a little score testing and some simple math will do the job nicely.

In Blackjack, bets are settled as follows:

If the Player's score is more than 21, he loses his bet.

If the Player's score is 21 or less and the Dealer's score is more than 21, he wins his bet.

If the Player's score is 21 or less and is more than the Dealer's score, he wins his bet.

If the dealer's score is 21 or less, and is more than the player's score, the player loses his bet.

If the Player's score is 21 or less and is the same as the Dealer's score, he keeps his bet (called a "push").

If the Player has Blackjack and the Dealer does not, he wins 1 1/2 times his bet.

We're getting near the end of our Course on BASIC Language programming. After you have finished it, there will be new programs to write and new techniques to learn — but you will be on your own. For practice, tackle this next challenging Assignment with minimum help. A detailed flow-chart (Figure 13-2) is included with the Assignment as a guide. You can do it!



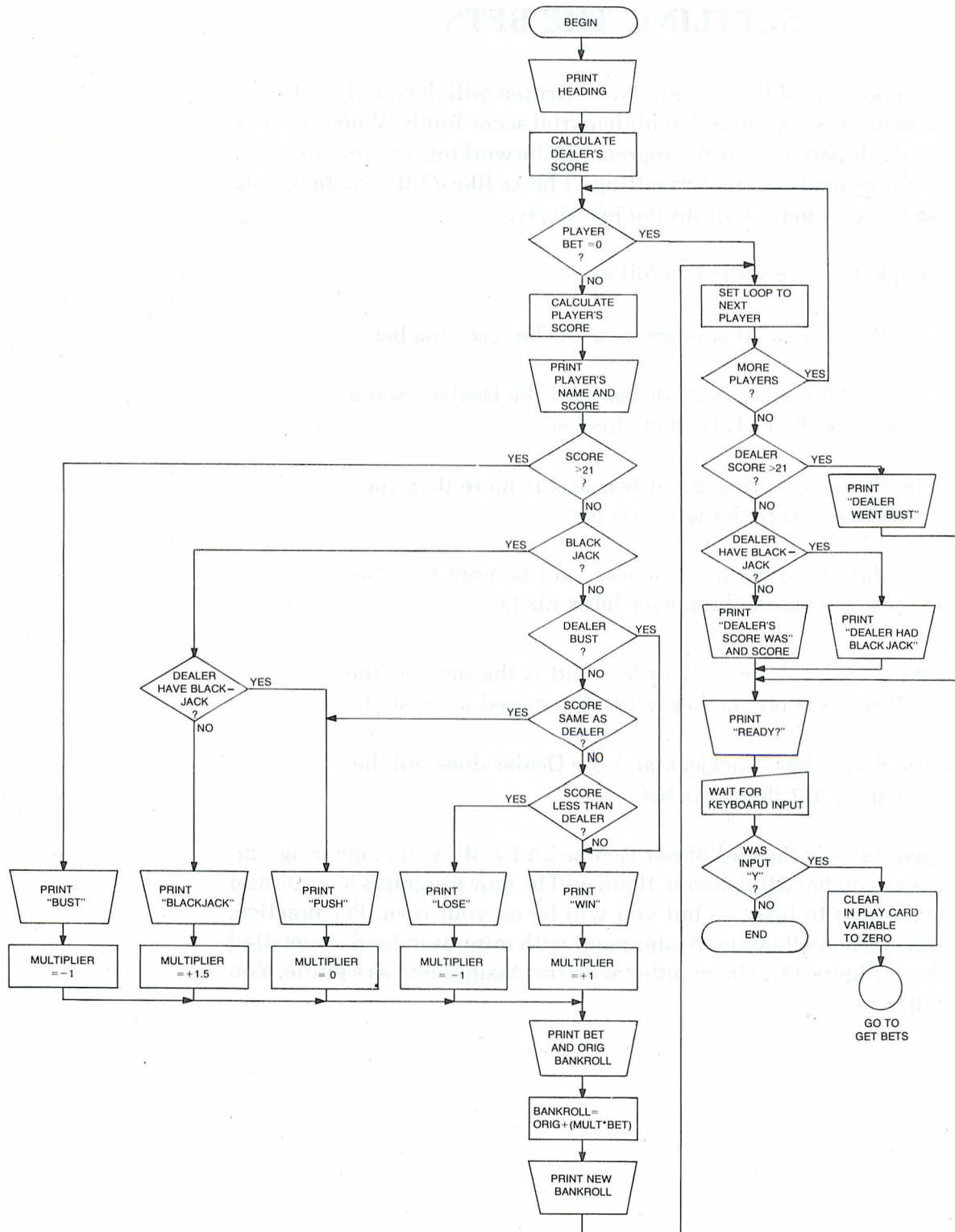


Figure 13-2

The settle bets flow-chart.

**ASSIGNMENT # 4 — Write a program to Settle Bets**

Write the program Statements that settle bets after each round of the game has been played. Bets are settled in accordance with the information just given. Use the following format for displaying the results of the hand:

**SETTLING BETS**

| PLAYER  | SCORE | RESULT | BET | ORIG BKRL | NEW BKRL |
|---------|-------|--------|-----|-----------|----------|
| WILLARD | 17    | LOSE   | 25  | 150       | 125      |
| LOUIS   | 20    | WIN    | 50  | 275       | 325      |
| RAINA   | 19    | PUSH   | 10  | 175       | 175      |
| BRUCE   | 24    | BUST   | 25  | 200       | 175      |
| DAVID   | 21    | BJACK  | 100 | 125       | 275      |
| WILLIAM | 22    | BUST   | 50  | 200       | 150      |
| RAY     | 20    | WIN    | 20  | 175       | 195      |

**DEALER'S SCORE WAS 19**

Suggestions: Calculate Dealer's absolute score before setting up the loop for Players; save it in variable D. In the body of the Player loop, calculate the Player's absolute score and save it in variable P. Using IF . . . THEN Statements, determine if the Player won, lost, pushed or had Blackjack. Based on this determination, establish a multiplier value in variable M. The multiplier is:

0 if push  
-1 if lose  
+1 if won  
+1.5 if Blackjack

To adjust a bankroll for win or loss:

$\langle \text{Bkrl} \rangle + (\langle \text{MULT} \rangle * \langle \text{Bet} \rangle)$  will do it

## EXERCISE B

### Load the Settle Bets. Test the Program

Load the saved program into your computer and add the "settle bets" program lines. Be sure to re-save the program after adding the new lines.

RUN the program and test it as thoroughly as possible. If a typing error has been made, BASIC will give you an error message that will provide a clue to what's wrong. If you have to make any corrections, be sure to re-save the program after all the changes have been made. When you are satisfied that everything works, read the following comments and continue to Segment 14.

Our step-by-step plan is almost complete; only the last step remains. We need to improve the formatting of the displayed information and add the fancy features; "gingerbread" as we call it.

The next Segment describes additions that were made to implement these new features, and also some interesting things we learned when we invited Mr. Watson over to show off our game. You remember Mr. Watson; the friend who does strange things "just to see what will happen!"



## ANSWERS

The correct answers are given below for the various questions asked in different parts of this Segment.

### Assignment # 1

PAGE 13-3

```
595 REM SET UP LOOP - CHECK FOR BET OF ZERO
600 FOR I=0 TO N:IF P(I,1)=0 GOTO 630
605 REM CHECK FOR BLACKJACK - TRY NEXT PLAYER IF SO
610 IF INT(P(I,4)/400)=1 GOTO 630
615 REM IF SCORE IS NOT BUST, GO PLAY DEALER'S HAND
620 IF FN M(P(I,4),100)<22 GOTO ***
625 REM LOOP TO POLL ALL PLAYERS
630 NEXT I
```

### Assignment # 2

PAGE 13-4

```
635 REM PRINT APPROPRIATE MESSAGE
640 PRINT "ALL PLAYERS HAVE BLACKJACK OR WENT BUST"
650 PRINT:PRINT "DEALER DOES NOT PLAY HIS HAND":GOTO ***
```

### Assignment # 3

PAGE 13-3

```
655 REM PRINT 'DEALER'S HAND' MESSAGE
660 PRINT "DEALER'S HAND":PRINT
665 REM DISPLAY DEALER'S ORIGINAL CARDS
670 PRINT "DEALER'S CARDS ARE: ";TAB(25)::C=D1:GOSUB 1090
680 PRINT TAB(25)::C=D2:GOSUB 1090
685 REM SCORE DEALER'S TWO CARDS
690 S=0:C=D1:GOSUB 1100:C=D2:GOSUB 1100
695 REM TEST SCORE FOR BLACKJACK - MARK IF SO
700 IF S=121 THEN S=421:GOTO 740
705 REM TEST SCORE FOR 17 OR MORE - GO PRINT SCORE IF SO
710 IF FN M(S,100)>16 GOTO 740
715 REM SCORE IS 16 OR LESS - DEAL ADDITIONAL CARD
720 GOSUB 1000
725 REM PRINT NEW CARD - SCORE - LOOP BACK
730 PRINT TAB(25)::GOSUB 1090:GOSUB 1100:GOTO 710
735 REM SAVE SCORE IN INFO FILE - PRINT SCORE MESSAGE
740 D3=S:PRINT
750 IF INT(S/400)=1 THEN PRINT "DEALER HAS BLACKJACK":GOTO 780
760 IF FN M(S,100)>21 THEN PRINT "DEALER WENT BUST":GOTO 780
770 PRINT "DEALER'S SCORE IS";FN M(S,100)
```

```
775 REM NEXT STATEMENT IS TEMPORARY FOR TESTING
780 STOP
```

## Assignment # 4

PAGE 13-11

```
775 REM PRINT HEADING FOR CHART
780 PRINT TAB(24);"SETTLING BETS":PRINT
790 PRINT "PLAYER SCORE RESULT BET ORIG BKRL NEW BKRL"
795 REM CALCULATE DEALER'S SCORE
800 D=FN M(D3,100)
805 REM SET UP LOOP FOR ALL PLAYERS
810 FOR I=0 TO N:IF P(I,1)=0 GOTO 940
815 REM CALCULATE PLAYER'S SCORE
820 P=FN M(P(I,4),100)
825 REM PRINT PLAYER'S NAME AND SCORE
830 PRINT P$(I);TAB(15);P;TAB(22);
835 REM PRINT APPROPRIATE MESSAGE - STORE MULTIPLIER
840 IF P>21 THEN PRINT "BUST";:M=-1:GOTO 920
845 REM CHECK PLAYER BLACKJACK
850 IF INT(P(I,4)/400)<>1 GOTO 880
855 REM DEALER HAVE BLACKJACK ALSO?
860 IF INT(D3/400)=1 GOTO 890
865 REM IF NOT, PLAYER WINS
870 PRINT "BJACK";:M=1.5:GOTO 920
875 REM CHECK IF DEALER BUST
880 IF D>21 GOTO 910
885 REM PLAYER SAME AS DEALER SCORE?
890 IF P=D THEN PRINT "PUSH";:M=0:GOTO 920
895 REM PLAYER LESS THAN DEALER SCORE?
900 IF P<D THEN PRINT "LOSE";:M=-1:GOTO 920
905 REM PLAYER WINS IF REACH HERE
910 PRINT "WIN";:M=1
915 REM PRINT BET AND OLD BANKROLL
920 PRINT TAB(30);P(I,1);TAB(38);P(I,0);TAB(49);
925 REM CALCULATE NEW BANKROLL AND PRINT
930 P(I,0)=P(I,0)+(M*P(I,1)):PRINT P(I,0)
935 REM LOOP FOR ALL PLAYERS
940 NEXT I
945 REM PRINT DEALER'S SCORE
950 PRINT:IF D>21 THEN PRINT "DEALER WENT BUST":GOTO 980
960 IF INT(D3/400)=1 THEN PRINT "DEALER HAD BLACKJACK":GOTO 980
970 PRINT "DEALER'S SCORE WAS";D
975 REM PAUSE FOR READING RESULTS - TYPE 'Y' TO CONTINUE
980 LINE INPUT "READY? ";X$:IF LEFT$(X$,1)<>"Y" THEN STOP
985 REM CLEAR CARDS IN PLAY - LOOP FOR ANOTHER HAND
990 C2=0:GOTO 100
```



# Individual Learning Program

## BASIC PROGRAMMING

### *Segment 14*

## GINGERBREAD AND BANDAGES

EC-1100

HEATH COMPANY  
BENTON HARBOR, MICHIGAN 49022

Copyright © 1977  
Heath Company  
All Rights Reserved  
Printed in the United States of America



## INTRODUCTION

The “Gingerbread” in the title of this Segment is for the fancy features and improvement in PRINTouts that our step-by-step plan lists as step 4 (see Page 11-4). The “Bandages” refers to repairs made to parts of the program which didn’t work as expected.

This is an important phase in program development. Now that the framework has been built and is functional, you can begin to add stained-glass windows to your cathedral to please the eye; carpets, and draperies for a sacramental hush, and statuary for excitement. It takes a sense of “showmanship”, but it is the part most people will notice first when you display your creation. Because of this, the effort will be worthwhile.

## IMPROVEMENTS

### ASSIGNMENT # 1 — Dealer Checks for Blackjack

Looking over the program objectives, we find that all of them have been met except one: If the Dealer's upcard is an Ace or one with a value of 10, he is to check his hole card to see if he has Blackjack before the game continues. We have left room in our program (lines 220 to 390) to add this feature.

Write the program lines that cause the Dealer to check for Blackjack if his upcard is an Ace or one with a value of 10. Create your own flow-chart to define exactly what needs to be done. The check should be made after the cards have been dealt and before any Players play out their hands. If the Dealer shows an Ace or 10-count, use the following PRINTouts as an objective:

(If Dealer does not have Blackjack):

```
DEALER SHOWS KING OF CLUBS
PEEKING AT DEALER'S HOLE CARD
(pause)
DEALER DOES NOT HAVE BLACKJACK
```

(Or, if Dealer has Blackjack);

```
DEALER SHOWS KING OF CLUBS
PEEKING AT DEALER'S HOLE CARD
(pause)
DEALER'S CARDS ARE: KING OF CLUBS
 ACE OF DIAMONDS
```

```
DEALER HAS BLACKJACK
```

Hint: One way to create a pause in a program is with a "do-nothing" loop. For example: FOR W=1 TO 600:NEXT W

If the Dealer has Blackjack, each Player's cards should be scored and the result saved in his Information File. This is needed so the "settling bets" portion of the program works properly.

If the Dealer has Blackjack, you can GOTO 670 in the "play Dealer's hand" part of the program after scoring the Players' cards.

Begin the check for Dealer Blackjack at line 220. The correct answer is on Page 14-18.

## TESTING THE PROGRAM

After adding the lines written in Assignment #1, we tested the program by playing a number of games. We made notes when something came up which we thought needed improvement. Here's our list:

1. Some Players might like to be notified when the cards have been shuffled.
2. When the two cards are being dealt to each Player and to the Dealer, there is a pause during which nothing appears to be happening. Find a way to indicate when each card is dealt.
3. Indent messages following a Player's name for better readability.
4. Indent messages following the "DEALER'S HAND" for the same reason.
5. Find a way to separate different parts of the game.
6. Find a way to separate each Player's hand.

Let's see what we can do to implement the improvements on our list.

## ASSIGNMENT # 2 — Indicate When Cards Are Shuffled

A "flag" is a known value stored in a variable to indicate that a specific event has occurred. At some point in a program, the flag can be tested to determine if something needs to be done. Whenever the deck of cards is shuffled, we can set up a flag to indicate the event and test the flag when we wish to PRINT an appropriate message. Using variable F as the flag variable, store the value 1 whenever the cards are shuffled. Test the flag at line 160 and if it is "set", PRINT the message:

CARDS HAVE BEEN SHUFFLED

After PRINTing the message, be sure to "reset" the flag to zero. Add a Statement at line 10 to "set" the flag so that the message will be PRINTed for the first hand of the game.

The correct answer is on Page 14-20.



**ASSIGNMENT # 3 — Indicate “Cards Being Dealt”**

The next item on the list of desired improvements calls for some indication that cards are being dealt, to occupy the pause that occurs during this part of the program. Add a feature to the program that PRINTs the following when the initial two cards are being dealt to the Players and the Dealer:

```
DEALING CARDS * * * * *
```

A “\*” symbol should be PRINTed as each card is dealt. This display should come after the “Cards have been shuffled” message. The correct answer is on Page 14-20.

**ASSIGNMENT # 4 — Indent Players’ and Dealer’s Hands**

When we see the PRINTout during the “get bets”, “play hands”, and “play Dealer’s hand” portions of the game, we think it would look better if the game information was indented from the Player’s name and the message “DEALER’S HAND”. Modify the program so that any game information following a Player’s name or the message “DEALER’S HAND” is indented two spaces. The modification would change this format:

```
WILLARD
```

```
YOUR BANKROLL IS $ 150
YOUR BET? $25
```

to this format:

```
WILLARD
```

```
YOUR BANKROLL IS $ 150
YOUR BET? $25
```

The numbers of the program lines to modify are in the Answers for “Assignment 4” . . . on Page 14-20.

NOTE: We also found that the PRINTout of the cards lined up better if we started them at position 22 on the line. Therefore, we suggest you change the TAB Statements in the following lines to TAB(22):

```
430 510 680
```

```
440 670 730
```

### ASSIGNMENT # 5 — Separating the Hands

To keep different segments of the program PRINTout and the individual Players' hands separated for better readability, they need to be divided somehow. Some CRT terminals will blank out the screen if sent a control code such as CTRL/Z. BASIC will transmit this "screen erase" code if you use the Statement print CHR\$(26).

We found that blanking the screen before each program segment made the best display, but not all CRT's have this feature. For those that don't, or where a printer is being used as the computer's console, a line drawn across the screen is next best.

Add a sub-program beginning at line 1170 to blank the screen if your CRT has this capability, or to draw a line if your computer console can't be erased on command. If your sub-program will blank the screen, add a do-nothing loop to pause for reading the information before blanking the screen. If you draw a line instead, add a do-nothing loop after drawing the line for a touch of showmanship.

The line should be a number of hyphens. How many depends on the number of characters on a line of screen printing or, in the case of a printer, not so many that it takes too long to draw.

Then, pick appropriate places in the program and insert GOSUB's to call on the sub-program.

The correct answer is on Page 14-20.

**ASSIGNMENT # 6 — A “Settling Bets” Correction**

It's time for the “bandages”.

After playing Blackjack with the computer for several hours, the only problem we could find was that if the Dealer did not play his hand because all Players either had Blackjack or had gone bust, the Dealer's score was still PRINTed when settling bets. Correct the “settling bets” portion of the program so the proper message is PRINTed if the Dealer did not play his hand because all Players either had Blackjack or went bust. The message should be:

DEALER DID NOT PLAY HIS HAND

Decide how to set up a flag to indicate that the Dealer did not play his hand, and add the appropriate Statement to the program.

Hint: In our “fix” for this problem, we did not use any new variables as a flag.

If you need to add a program line, do not use a number ending in 5, since we have reserved such numbers for REMark lines. The correct answer is on Page 14-21.



## WATSON'S TEST GAME

After adding the "bandage" described in Assignment 6, we again played Blackjack with the computer for several hours. Proud, but weary, and confident that each and every bug in the program had been taken care of, we invited our friend Mr. Watson over to show off our new creation. He didn't let us down!

After we explained how the game was played, Watson sat down at the keyboard. Here's a PRINTout of part of the very first game we played:

-----  
PLAYER'S NAME? DALE  
PLAYER'S NAME? WILLARD  
PLAYER'S NAME? Q  
-----

DALE

YOUR BANKROLL IS \$ 150  
YOUR BET? \$25.2938455029940298332

You can imagine the effect such a bet had on the formatting of the PRINTout in the "settling bets" part of the program. We made a note to fix that problem. Then we told Watson not to do that any more and started a new game. He grinned and typed RUN.

-----  
PLAYER'S NAME? DALEAWATSONNOWISTHETIMEFORALLGOODMENTOCOMETOTHEAIDOF  
THEIRCOUNTRYTHEQUICKBROWNFOXJUMPEDOVERTHELAZYDOG'SBACK  
-----

We wrote down our second note and told Watson not to do that any more, either!

-----  
PLAYER'S NAME? DALE  
PLAYER'S NAME? WILLARD  
PLAYER'S NAME? Q  
-----

DALE

YOUR BANKROLL IS \$ 150

YOUR BET? \$-25

Note number three. Please, Watson, bet only positive amounts. You could see that he was enjoying our new game program immensely.

PLAYER'S NAME? DALE

PLAYER'S NAME? WILLARD

PLAYER'S NAME? 0

DALE

YOUR BANKROLL IS \$ 150

YOUR BET? \$150

WILLARD

YOUR BANKROLL IS \$150

YOUR BET? \$25

CARDS HAVE BEEN SHUFFLED

DEALING CARDS \* \* \* \* \*

DALE

DEALER SHOWS FIVE OF CLUBS

YOUR CARDS ARE: FOUR OF HEARTS

NINE OF DIAMONDS

HIT? Y

JACK OF CLUBS

YOU WENT BUST

---

WILLARD

DEALER SHOWS FIVE OF CLUBS

YOUR CARDS ARE: SIX OF SPADES

QUEEN OF SPADES

HIT? Y

SEVEN OF CLUBS

YOU WENT BUST

---

ALL PLAYERS HAVE BLACKJACK OR WENT BUST

DEALER DOES NOT PLAY HIS HAND

---

SETTLING BETS

| PLAYER  | SCORE | RESULT | BET | ORIG BKRL | NEW BKRL |
|---------|-------|--------|-----|-----------|----------|
| DALE    | 23    | BUST   | 150 | 150       | 0        |
| WILLARD | 23    | BUST   | 25  | 150       | 125      |

DEALER DID NOT PLAY HIS HAND

---

READY? Y

---

DALE

YOUR BANKROLL IS \$ 0

YOUR BET? \$ 0

Note number four! The Player has no bankroll left but he is still asked for a bet. We thanked Watson and sent him home. He said that he would be glad to come back and help any time. Here are the required "anti-Watsons", according to the notes we made:

1. Limit bets to two decimal places.
2. Limit names to 10 characters maximum.
3. Limit bets to positive values.
4. Omit Players who have bankrolls of zero.



## MAKING REPAIRS

### ASSIGNMENT # 7 — Positive-Value Bets

Try your hand at constructing a complex mathematical expression that will convert a Player's bet to a **positive value** with a maximum of **two decimal places**. Review Segment 5 for information about BASIC functions such as ABS(X) and INT(X).

The objective is to add a Statement to line 130 which will assure that the amount stored in the Player's Information File as his bet is positive and has no more than two decimal places. That will take care of anti-Watsons numbers 1 and 3. Remember that the function ABS(X) can be used to make a positive value out of a negative one, and INT(X) discards any decimal portion of a value. You might consider using INT(X) after multiplying to obtain the desired number of decimal place.

The correct answer is on Page 14-21.

### ASSIGNMENT # 8 — Limit Player's Names to 10 Characters

Add the necessary Statement to line 1050 of the program so that the name stored in the Player's Information File is no more than 10 characters long. That will take care of anti-Watson #2.

HINT: Left\$

The correct answer is on Page 14-21.

### ASSIGNMENT # 9 — Exclude Players With No Bankroll

The last anti-Watson needs to be added so Players who have exhausted their bankroll are excluded from the game. Find a way to exclude them — Be clever. The solution we have in mind can accomplish this Assignment by adding only two Statements to the program, both of them at line 100. Remember, we have already included Statements in the program to omit Player's who have bet zero.

The correct answer is on Page 14-21.

**ASSIGNMENT # 10 — Poll All Player's Bankrolls**

After making all the changes for gingerbread and bandages and anti-Watsons, we again tested the program and found one new problem: If all the Players have exhausted their bankrolls, the Dealer begins to play himself; we have fixed it so the Players with zero bankrolls are excluded. So here is your final Assignment!

Add Statements at line 990 that will poll all Players' bankrolls to see if anyone has any money left. If even one of the Players has a bankroll greater than zero, the program should GOTO line 100 for another round of play. If no Player has a bankroll greater than zero, the program should GOTO line 90 to get new names and start over. If a program line must be added, number it 992.

The correct answer is on Page 14-21.

## PROGRAM COMPLETED

Our Course on BASIC Language Computer Programming is complete; it now meets all the objectives we set out to accomplish.

You have been introduced to all the standard programming Statements of the language, performed Assignments that gave practical examples of the use of the Statements, performed Exercises that gave you practice in using the Statements, and participated in the creation of a reasonably extensive program that illustrated a step-by-step method of program writing.

From this point on, it's practice and more practice that will lead you to the creation of your monument. For starters, we have a few suggestions that you may wish to try adding to your Blackjack program. We know they can be done because we have done them. The program we show to our friends has all the following features:

1. Permit a Player to leave the game permanently by betting "X".
2. Allow new Players to be brought into the game without affecting the bankrolls of the other players. This feature is initiated when a Player bets "N". (This is why we implemented the "get names" portion as a sub-program.)
3. Allow Players to split pairs — several times if desired.
4. Allow Players to "double down" if desired. (See "Appendix A" for the rules on doubling down.)
5. Allow Players to make "insurance" bets if the Dealer's upcard is an Ace.

These features and more can be added to enhance your program. You are only limited by your imagination, and the amount of memory available in your computer.



In the “real world” of computers, one sometimes encounters a brick wall for which BASIC has a disarmingly simple message: ! ERROR — MEM OVERFLO, or on other computers perhaps, OUT OF MEMORY. Whatever form the message takes, its meaning will be clear: Your program is too long for the amount of memory circuits installed in your computer.

In personal, or “hobby” computers, this message is seen all too often. So what can you do if BASIC says MEM OVERFLO? There are only two alternatives. You may purchase more memory or you may shorten your program.

If our Blackjack program has gotten so large that your computer said MEM OVERFLO and you decide to shorten it, we suggest that you attack the strings. The messages, card names, suits, and other strings in this or any program take more than their share of memory. The more of these you can leave out or shorten, the greater the effect on reducing memory usage.

Thanks for taking our Course. Good Luck, and Good Programming!

## THE COMPLETE BLACKJACK PROGRAM

A complete Listing of the Blackjack program begins on Page 14-16. Some program lines have been changed from those developed in the Course. The reason is that the Course developed each part of the program in a step-by-step manner so that it was easier to understand. After all of the program was written, there were places where improvements could be made to reduce the number of Statements needed.

This "cleaning up" of a program is quite customary. Here's what we did:

1. Combined short program lines into longer lines with multiple Statements. Memory usage is less for lines with multiple Statements than it is for the same Statements spread over several program lines. Program execution is faster too, but that is not a problem in our game program. Care must be taken when you are combining lines. For example, lines 580 and 590 can't be combined because 590 is the target for a GOTO if the Player has bet zero. If these lines were combined, extra lines would be drawn across the screen for Players who were sitting out the hand. Compare the line numbers in the Listing with those of the program we developed in the Course to see which ones we combined.
2. We made a rather extensive change when we discovered that there were two parts of the program where the Player's two initial cards were scored; one if the Dealer's initial cards scored Blackjack, and one in the "Player plays out his hand" part. We decided to score the Player's cards at the time they were dealt and eliminate the duplication of Statements in the other two parts. The changes we made involve line numbers 170, 190, 260, 290, 300, 440, 450, and 470 (before we combined them, as discussed in paragraph 1, above).
3. We found the Statement

```
PRINT "DEALER SHOWS " ; : C=D1 : GOSUB 1090
```

appearing twice in the program (line 230 and 420). We decided to implement this Statement group as a sub-program at line 1190 and use GOSUBs at lines 230 and 410.

4. We found that the do-nothing loop at line 1180 which is part of the line drawing sub-program could be used as a target for a GOSUB. This allowed the duplicated do-nothing loop at line 250 to be replaced with a GOSUB 1180 at the end of line 230.

Careful evaluation of your completed program for changes, such as those we listed above, will result in a more compact program with less memory usage, faster execution, and a more "professional" appearance.

**Final Program Listing:**

```

10 GOSUB 1170: DIM D(51), N$(12), S$(3), P$(6), P(6,4): C1=52: F=1
20 FOR I=0 TO 51: D(I)=I: NEXT I
30 FOR I=0 TO 12: READ N$(I): NEXT I
40 FOR I=0 TO 3: READ S$(I): NEXT I
50 DATA "ACE", "TWO", "THREE", "FOUR", "FIVE", "SIX", "SEVEN"
60 DATA "EIGHT", "NINE", "TEN", "JACK", "QUEEN", "KING"
70 DATA "DIAMONDS", "CLUBS", "HEARTS", "SPADES"
80 DEF FN M(M1,M2)=M1-(M2*INT(M1/M2))
90 GOSUB 1040: IF N<0 GOTO 90
100 GOSUB 1170: FOR I=0 TO N: IF P(I,0)<=0 THEN P(I,1)=0: GOTO 160
110 PRINT P$(I): PRINT: PRINT " YOUR BANKROLL IS $"; P(I,0)
130 INPUT " YOUR BET? $"; X: P(I,1)=ABS((INT(X*100))/100)
140 IF P(I,1)<=P(I,0) THEN PRINT: GOTO 160
150 PRINT " BET IS MORE THAN BANKROLL - NO CREDIT!": GOTO 130
160 NEXT I: GOSUB 1170: IF F=1 THEN PRINT "CARDS HAVE BEEN SHUFFLED": F=
170 PRINT "DEALING CARDS";: FOR D=0 TO N: S=0: IF P(D,1)=0 GOTO 200
190 FOR L=2 TO 3: GOSUB 1000: P(D,L)=C: GOSUB 1100: PRINT " *";: NEXT L
196 IF S=121 THEN S=421
198 P(D,4)=S
200 NEXT D: GOSUB 1000: D1=C: PRINT " *";: GOSUB 1000: D2=C: PRINT " *"
220 GOSUB 1170: S=0: C=D1: GOSUB 1100: IF S<10 GOTO 400
230 GOSUB 1190: PRINT " PEEKING AT DEALER'S HOLE CARD": GOSUB 1180
260 C=D2: GOSUB 1100: IF S=121 THEN PRINT: GOTO 670
270 PRINT " DEALER DOES NOT HAVE BLACKJACK": GOSUB 1170
400 FOR P=0 TO N: IF P(P,1)=0 GOTO 590
410 PRINT P$(P): PRINT: GOSUB 1190: PRINT
430 PRINT " YOUR CARDS ARE:"; TAB(22);: C=P(P,2): GOSUB 1090
440 C=P(P,3): PRINT TAB(22);: GOSUB 1090: S=P(P,4)
470 IF S=421 GOTO 540
480 LINE INPUT " HIT? "; H$: IF LEFT$(H$,1)<>"Y" GOTO 540
500 GOSUB 1000: PRINT TAB(22);: GOSUB 1090: GOSUB 1100: P(P,4)=S
530 IF FN M(S,100)<21 GOTO 480
540 PRINT
550 IF INT(S/400)=1 THEN PRINT " YOU HAVE BLACKJACK": GOTO 580
560 IF FN M(S,100)>21 THEN PRINT " YOU WENT BUST": GOTO 580
570 PRINT " YOUR SCORE IS": FN M(S,100)
580 GOSUB 1170
590 NEXT P
600 FOR I=0 TO N: IF P(I,1)=0 GOTO 630
610 IF INT(P(I,4)/400)=1 GOTO 630
620 IF FN M(P(I,4),100)<22 GOTO 660
630 NEXT I
640 PRINT "ALL PLAYERS HAVE BLACKJACK OR WENT BUST"
650 PRINT: PRINT "DEALER DOES NOT PLAY HIS HAND": D3=0: GOTO 780
660 PRINT "DEALER'S HAND": PRINT
670 PRINT " DEALER'S CARDS ARE:"; TAB(22);: C=D1: GOSUB 1090
680 PRINT TAB(22);: C=D2: GOSUB 1090
690 S=0: C=D1: GOSUB 1100: C=D2: GOSUB 1100: D3=S

```



```

700 IF S=121 THEN S=421:D3=S:GOTO 740
710 IF FN M(S,100)>16 GOTO 740
720 GOSUB 1000
730 PRINT TAB(22);:GOSUB 1090:GOSUB 1100:D3=S:GOTO 710
740 PRINT
750 IF INT(S/400)=1 THEN PRINT " DEALER HAS BLACKJACK :GOTO 780
760 IF FN M(S,100)>21 THEN PRINT " DEALER WENT BUST":GOTO 780
770 PRINT " DEALER'S SCORE IS";FN M(S,100)
780 GOSUB 1170:PRINT TAB(24);"SETTLING BETS":PRINT
790 PRINT "PLAYER SCORE RESULT BET ORIG BKRL NEW BKRL"
800 D=FN M(D3,100)
810 FOR I=0 TO N:IF P(I,1)=0 GOTO 940
820 P=FN M(P(I,4),100):PRINT P$(I);TAB(15);P;TAB(22);
840 IF P>21 THEN PRINT "BUST";:M=-1:GOTO 920
850 IF INT(P(I,4)/400)<1 GOTO 880
860 IF INT(D3/400)=1 GOTO 890
870 PRINT "BJACK";:M=1.5:GOTO 920
880 IF D>21 GOTO 910
890 IF P=D THEN PRINT "PUSH";:M=0:GOTO 920
900 IF P<D THEN PRINT "LOSE";:M=-1:GOTO 920
910 PRINT "WIN";:M=1
920 PRINT TAB(30);P(I,1);TAB(38);P(I,0);TAB(49);
930 P(I,0)=P(I,0)+(M*P(I,1)):PRINT P(I,0)
940 NEXT I
950 PRINT:IF D>21 THEN PRINT "DEALER WENT BUST":GOTO 980
958 IF D=0 THEN PRINT "DEALER DID NOT PLAY HIS HAND":GOTO 980
960 IF INT(D3/400)=1 THEN PRINT "DEALER HAD BLACKJACK":GOTO 980
970 PRINT "DEALER'S SCORE WAS";D
980 GOSUB 1170:LINE INPUT "READY? ";X$:IF LEFT$(X$,1)<>"Y" THEN STOP
990 C2=0:FOR I=0 TO N:IF P(I,0)>0 GOTO 100
992 NEXT I:GOTO 90
1000 X=INT(C1*RND(1)):C=D(X)
1010 FOR I=X TO 50:D(I)=D(I+1):NEXT I
1020 D(51)=C:C1=C1-1:C2=C2+1:IF C1=1 THEN C1=52-C2:F=1
1030 RETURN
1040 N=-1
1050 LINE INPUT "PLAYER'S NAME? ";X$:P$(N+1)=LEFT$(X$,10)
1060 IF X$="0" THEN RETURN
1070 N=N+1:P(N,0)=150:IF N<6 GOTO 1050
1080 PRINT "SEVEN PLAYERS MAXIMUM":RETURN
1090 PRINT N$(FN M(C,13));" OF ";S$(FN M(C,4)):RETURN
1100 S1=FN M(C,13)+1:IF S1>10 THEN S1=10
1110 IF INT(S/100)=1 GOTO 1140
1120 IF S1<>1 GOTO 1140
1130 S1=111
1140 S=S+S1:IF FN M(S,100)<=21 THEN RETURN
1150 IF INT(S/100)<>1 THEN RETURN
1160 S=S-110:RETURN
1170 FOR W=0 TO 62:PRINT "-";:NEXT W:PRINT
1180 FOR W=1 TO 600:NEXT W:RETURN
1190 PRINT " DEALER SHOWS ";:C=D1:GOTO 1090

```

## ANSWERS

The correct answers are given below for the various questions asked in different parts of this Segment.

### Assignment # 1

PAGE 14-3

Match the flow-chart you drew to Figure 14-1, the "Flow-chart for assignment #1." They should be very similar.

The following program lines show the way we implemented our flow-chart. If your program lines differ, they are "correct" if they accomplish the objective.

```

215 REM DEALER'S UPCARD ACE OR 10-COUNT? S=111 OR 10 IF SO
220 S=0:C=D1:GOSUB 1100:IF S<10 GOTO 400
225 REM BLACKJACK POSSIBLE - PRINT MESSAGE
230 PRINT "DEALER SHOWS ";;GOSUB 1090
240 PRINT "PEEKING AT DEALER'S HOLE CARD"
245 REM DO NOTHING LOOP TO SIMULATE PEEKING
250 FOR W=1 TO 600:NEXT W
255 REM CHECK IF BLACKJACK
260 C=D2:GOSUB 1100:IF S=121 GOTO 280
265 REM NO BLACKJACK - PRINT MESSAGE - ON WITH THE HAND
270 PRINT "DEALER DOES NOT HAVE BLACKJACK":GOTO 400
275 REM DEALER HAS BLACKJACK - SCORE PLAYERS' CARDS
280 FOR I=0 TO N:IF P(I,1)=0 GOTO 300
290 S=0:C=P(I,2):GOSUB 1100:C=P(I,3):GOSUB 1100:P(I,4)=S
295 REM SCORE ALL PLAYERS THEN GOTO PLAY DEALER'S HAND
300 NEXT I:GOTO 670

```

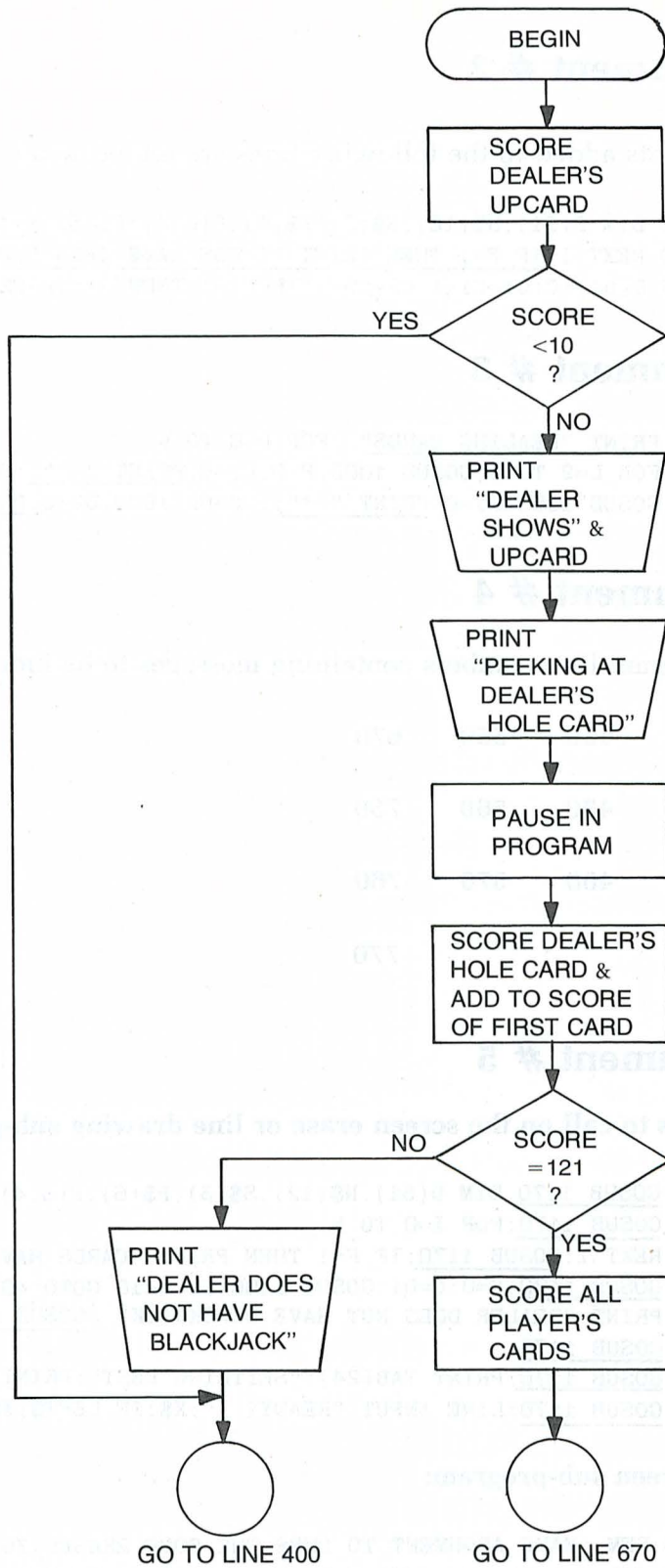


Figure 14-1  
Flow chart for Assignment 1.



## Assignment # 2

PAGE 14-4

(Statements added to the following lines are underlined.)

```

10 DIM D(51),N$(12),S$(3),P$(6),P(6,4):C1=52:F=1
160 NEXT I:IF F=1 THEN PRINT "CARDS HAVE BEEN SHUFFLED":F=0
1020 D(51)=C:C1=C1-1:C2=C2+1:IF C1=1 THEN C1=52-C2:F=1

```

## Assignment # 3

PAGE 14-5

```

170 PRINT "DEALING CARDS";:FOR D=0 TO N
190 FOR L=2 TO 3:GOSUB 1000:P(D,L)=C:PRINT " *";:NEXT L
210 GOSUB 1000:D1=C:PRINT " *";:GOSUB 1000:D2=C:PRINT " *"

```

## Assignment # 4

PAGE 14-6

The program line numbers containing messages to be indented are:

```

120 420 550 670
130 430 560 750
150 480 570 760
 770

```

## Assignment # 5

PAGE 14-6

GOSUB's to call on the screen erase or line drawing sub-program:

```

10 GOSUB 1170:DIM D(51),N$(12),S$(3),P$(6),P(6,4):C1=52:F=1
100 GOSUB 1170:FOR I=0 TO N
160 NEXT I:GOSUB 1170:IF F=1 THEN PRINT "CARDS HAVE BEEN SHUFFLED":F=0
220 GOSUB 1170:S=0:C=D1:GOSUB 1100:IF S<10 GOTO 400
270 PRINT "DEALER DOES NOT HAVE BLACKJACK" :GOSUB 1170:GOTO 400
580 GOSUB 1170
780 GOSUB 1170:PRINT TAB(24);"SETTLING BEST":PRINT
980 GOSUB 1170:LINE INPUT "READY? ";X$:IF LEFT$(X$,1)<>"Y THEN STOP

```

Erase screen sub-program:

```

1165 REM MAKE ARGUMENT TO CHR$ THE CODE ERASES YOUR SCREEN
1170 FOR W=1 TO 600:NEXT W:PRINT CHR$(26);:RETURN

```

Draw a line sub-program:

```
1165 REM MAKE LENGTH OF LINE FIT YOUR SCREEN OR PRINTER
1170 FOR W=0 TO 62:PRINT "-";:NEXT W:PRINT
1175 REM DO-NOTHING PAUSE LOOP
1180 FOR W=1 TO 600:NEXT W:RETURN
```

## Assignment # 6

PAGE 14-7

Here's our "fix" for the problem. Any solution you developed that works is OK!

```
645 REM FLAG CONSISTS OF SETTING DEALER'S SCORE TO ZERO
650 PRINT:PRINT "DEALER DOES NOT PLAY HIS HAND":D3=0:GOTO 780
955 REM PRINT MESSAGE IF DEALER'S SCORE IS ZERO
958 IF D=0 THEN PRINT "DEALER DID NOT PLAY HIS HAND":GOTO 980
```

## Assignment # 7

PAGE 14-11

```
130 INPUT " YOUR BET? $";X:P(I,1)=ABS((INT(X*100))/100)
```

## Assignment # 8

PAGE 14-11

```
1050 LINE INPUT "PLAYER'S NAME? ";X$:P$(N+1)=LEFT$(X$,10)
```

## Assignment # 9

PAGE 14-11

```
100 GOSUB 1170:FOR I=0 TO N:IF P(I,0)<=0 THEN P(I,1)=0:GOTO 160
```

By setting the Player's bet to zero if he has no bankroll, all the other parts of the program will leave him out.

## Assignment # 10

PAGE 14-12

```
990 C2=0:FOR I=0 TO N:IF P(I,0)>0 GOTO 100
992 NEXT I:GOTO 90
```





*Part III*

**Appendixes**

Appendix A

## APPENDIX A

### Description and Playing Rules for Blackjack

1. Use a standard deck of playing cards.
2. "Count" each card at its face value, with these exceptions: Jacks, Queens and Kings all count 10. Aces count at 1 or 11 at the Player's option.
3. The object of the game is to achieve a higher count of cards than that of the Dealer without exceeding 21.
4. At the beginning of a round of play, each Player places a bet on the outcome. (In our game we won't use money. The computer will give you a starting bankroll and keep track of your wins and losses.)
5. The computer is the permanent Dealer and Banker of the game.
6. A Player loses his bet the moment his card count exceeds 21 (called "going Bust").
7. The Dealer deals two cards face down to each Player, one card to himself face down (called his "hole card") and one card to himself face up (called his "upcard").



8. All Players can see the Dealer's upcard.
9. When all cards are dealt, each Player in turn looks at his two cards and decides if he is satisfied with his score and wishes no more cards (he "stands"), or if he wishes to draw additional cards in an attempt to better his score (he "takes a hit").
10. If the initial two cards dealt to a Player count 21, it is called "Blackjack".
11. When a Player is satisfied with his hand and "stands", the next Player plays out his hand.
12. When all Players have played out their hands, it is the Dealer's turn. The Dealer plays his hand in the same way, with the exception that his strategy is fixed and is known to all Players. The Dealer must take an additional card when his count is 16 or less, and must stand on a count of 17 or more.
13. When the Dealer has stood or gone Bust, the bets are settled in accordance with the following Chart:

#### DISPOSITION OF BETS

| S c o r e s  |                   |                    |
|--------------|-------------------|--------------------|
| Player's     | Dealer's          | Disposition of Bet |
| More than 21 | Any               | Player loses       |
| 21 or less   | Less than Player  | Player wins        |
| 21 or less   | More than 21      | Player wins        |
| 21 or less   | More than Player  | Player loses       |
| 21 or less   | Same as Player    | Player keeps bet   |
| Blackjack    | Any-not Blackjack | Player wins 1-1/2  |
| Blackjack    | Blackjack         | Player keeps bet   |

**OPTIONS AVAILABLE TO PLAYERS (but not to the Dealer)**

**Double Down** If, after looking at his initial two cards, a Player feels he has an advantage, he may elect to double his bet and receive a single additional card.

**Split** A Player whose initial two cards are of the same value (called a "pair") may split them into two hands by doubling his bet to cover the second hand. Each hand is played individually as described above for single hands. Note that two face cards, or a face card and a Ten, may be split since they all count 10. When Aces are split, the Player receives only one additional card on each hand. If any additional cards dealt to split hands cause the count to be exactly 21, it does not count as Blackjack.

**Insurance** When the Dealer's upcard is an Ace, Players are offered the opportunity of buying insurance against the Dealer's having Blackjack. Insurance costs one half of the original bet. After all Players have decided whether or not to buy insurance, the Dealer peeks at his hole card. If it is a Ten count, bets are settled immediately as shown in the following Chart. If the Dealer's hole card is not a Ten count, all insurance bets are collected and the hand continues.

**DISPOSITION OF BETS WHEN DEALER HAS BLACKJACK**
**AFTER INSURANCE ROUND**

| Player's Score | Insured? | Disposition of Bet |
|----------------|----------|--------------------|
| Less than 21   | No       | Player loses       |
| Less than 21   | Yes      | Player keeps bet   |
| Blackjack      | No       | Player keeps bet   |
| Blackjack      | Yes      | Player keeps bet   |

**ADDITIONAL RULES OF PLAY:**

14. If the Dealer's upcard is a Ten count, he peeks at his hole card before any Players play out their hand. If the Dealer's hole card is an Ace, bets are settled immediately in accordance with rule 13 above. If the hole card is not an Ace, play continues.
15. If all Players' hands are either bust or Blackjack, the Dealer does not play out his hand.

Note that some rules that are popular in Blackjack games played at home are not included in the rules for Casino play. For example, having a count of 21 or less with five cards is not acknowledged as an automatic win. Also, the cards are not shuffled after each Blackjack hand turns up as is sometimes specified for home games.

| Player's Hand    | Dealer's Hand | Result       |
|------------------|---------------|--------------|
| Player busts     | Any           | Player loses |
| Player keeps bet | 21            | Player wins  |
| Player keeps bet | 20            | Player wins  |
| Player keeps bet | 19            | Player wins  |
| Player keeps bet | 18            | Player wins  |
| Player keeps bet | 17            | Player wins  |
| Player keeps bet | 16            | Player wins  |
| Player keeps bet | 15            | Player wins  |
| Player keeps bet | 14            | Player wins  |
| Player keeps bet | 13            | Player wins  |
| Player keeps bet | 12            | Player wins  |
| Player keeps bet | 11            | Player wins  |
| Player keeps bet | 10            | Player wins  |
| Player keeps bet | 9             | Player wins  |
| Player keeps bet | 8             | Player wins  |
| Player keeps bet | 7             | Player wins  |
| Player keeps bet | 6             | Player wins  |
| Player keeps bet | 5             | Player wins  |
| Player keeps bet | 4             | Player wins  |
| Player keeps bet | 3             | Player wins  |
| Player keeps bet | 2             | Player wins  |
| Player keeps bet | 1             | Player wins  |



## **APPENDIX B**

### **Microprocessors**

#### **(Number Systems and Codes)**

This Appendix is a portion of a Heathkit Individual Learning Program on microprocessors. It explains in detail both the Binary numbering system and the other numbering systems in common use.

You may find it useful to consider this Appendix as part of the course in Basic Programming, and perform its exercises and Self-Test Reviews.

## CONTENTS

|                                 |       |
|---------------------------------|-------|
| Introduction .....              | 16-3  |
| Unit Objectives .....           | 16-4  |
| Unit Activity Guide .....       | 16-5  |
| Decimal Number System .....     | 16-6  |
| Binary Number System .....      | 16-11 |
| Octal Number System .....       | 16-20 |
| Hexadecimal Number System ..... | 16-29 |
| Binary Codes .....              | 16-42 |
| Experiments 1 and 2 .....       | 16-58 |
| Unit Examination .....          | 16-59 |
| Examination Answers .....       | 16-61 |

## Unit 1

# NUMBER SYSTEMS AND CODES

## INTRODUCTION

The purpose of this first unit on microprocessors is to give you a firm foundation in number systems and codes. Binary numbers and codes are the basic language of all microprocessors. Octal and hexadecimal numbers allow easy manipulation of binary numbers and data. Thus, a good foundation in numbers and codes is essential to understanding microprocessors.

This unit will reacquaint you with the decimal number system, then expand the basic concept of numbers to the binary, octal, and hexadecimal systems. Understanding these systems fully will help you understand the many digital codes used with microprocessors. Although this unit can only give you a working knowledge of numbers and codes, you will become more proficient with them as you proceed through the units that follow.

A listing of number system tables has been provided in Appendix B of this course. Appendix B is located at the back of the second binder.

Examine the Unit Objectives listed in the next section to see what you will learn in this unit. Then follow the instructions in the Unit Activity Guide to be sure you perform all of the steps necessary to complete this lesson successfully. Check off each step as you complete it and, in the spaces provided, keep track of the time you spend on each activity.



## UNIT OBJECTIVES

When you complete this unit you will have the following knowledge and capabilities:

1. Given any decimal number, you will be able to convert it into its binary, octal, hexadecimal, and BCD equivalent.
2. Given any binary number, you will be able to convert it into its decimal, octal, hexadecimal, and BCD equivalent.
3. Given any octal number, you will be able to convert it into its decimal and binary equivalent.
4. Given any hexadecimal number, you will be able to convert it into its decimal and binary equivalent.
5. Given any BCD code, you will be able to convert it into its decimal and binary equivalent.
6. Given a list of popular digital codes, you will be able to read and identify them including pure binary, natural 8421 BCD, Gray, ASCII, and BAUDOT.
7. You will be able to convert a letter or number into its ASCII binary code, and convert an ASCII binary code into its letter or number equivalent.
8. You will be able to define the following terms:

|             |                               |
|-------------|-------------------------------|
| Radix       | BCD                           |
| Integer     | Gray Code                     |
| Decimal     | ASCII                         |
| Binary      | BAUDOT                        |
| Octal       | Most Significant Bit (MSB)    |
| Hexadecimal | Least Significant Bit (LSB)   |
| Bit         | Most Significant Digit (MSD)  |
| Parity      | Least Significant Digit (LSD) |

## UNIT ACTIVITY GUIDE

|                                                                     | Completion<br>Time |
|---------------------------------------------------------------------|--------------------|
| <input type="checkbox"/> Read section on Decimal Number System.     | _____              |
| <input type="checkbox"/> Answer Self-Test Review questions 1 — 6.   | _____              |
| <input type="checkbox"/> Read section on Binary Number System.      | _____              |
| <input type="checkbox"/> Answer Self-Test Review questions 7 — 13.  | _____              |
| <input type="checkbox"/> Read section on Octal Number System.       | _____              |
| <input type="checkbox"/> Answer Self-Test Review questions 14 — 19. | _____              |
| <input type="checkbox"/> Read section on Hexadecimal Number System. | _____              |
| <input type="checkbox"/> Answer Self-Test Review questions 20 — 25. | _____              |
| <input type="checkbox"/> Read section on Binary Codes.              | _____              |
| <input type="checkbox"/> Answer Self-Test Review questions 26 — 36. | _____              |
| <input type="checkbox"/> Perform Experiments 1 and 2.               | _____              |
| <input type="checkbox"/> Complete the Unit Examination.             | _____              |
| <input type="checkbox"/> Check Examination Answers.                 | _____              |

## DECIMAL NUMBER SYSTEM

The number system we are all familiar with is the decimal number system. This system was originally devised by Hindu mathematicians in India about 400 A.D. The Arabs began to use the system about 800 A.D., where it became known as the Arabic Number System. After it was introduced to the European community about 1200 A.D., the system soon acquired the title “decimal number system.”

A basic distinguishing feature of a number system is its **base** or **radix**. The base indicates the number of characters or digits used to represent quantities in that number system. The decimal number system has a base or radix of 10 because we use the ten digits 0 through 9 to represent quantities. When a number system is used where the base is not known, a subscript is used to show the base. For example, the number  $4603_{10}$  is derived from a number system with a base of 10.

**Positional Notation** The decimal number system is positional or weighted. This means each digit position in a number carries a particular weight which determines the magnitude of that number. Each position has a weight determined by some power of the number system base, in this case 10. The positional weights are  $10^0$  (units)\*,  $10^1$  (tens),  $10^2$  (hundreds), etc. Refer to Figure 1-1 for a condensed listing of powers of 10.

|        |                 |
|--------|-----------------|
| $10^0$ | = 1             |
| $10^1$ | = 10            |
| $10^2$ | = 100           |
| $10^3$ | = 1,000         |
| $10^4$ | = 10,000        |
| $10^5$ | = 100,000       |
| $10^6$ | = 1,000,000     |
| $10^7$ | = 10,000,000    |
| $10^8$ | = 100,000,000   |
| $10^9$ | = 1,000,000,000 |

Figure 1-1

Condensed listing of powers of 10.

\*Any number with an exponent of zero is equal to one.



We evaluate the total quantity of a number by considering the specific digits and the weights of their positions. For example, the decimal number 4603 is written in the shorthand notation with which we are all familiar. This number can also be expressed with positional notation.

$$\begin{aligned}(4 \times 10^3) + (6 \times 10^2) + (0 \times 10^1) + (3 \times 10^0) &= \\(4 \times 1000) + (6 \times 100) + (0 \times 10) + (3 \times 1) &= \\4000 + 600 + 0 + 3 &= 4603_{10}\end{aligned}$$

To determine the value of a number, multiply each digit by the weight of its position and add the results.

**Fractional Numbers** So far, only **integer** or whole numbers have been discussed. An integer is any of the natural numbers, the negatives of these numbers, or zero (that is, 0, 1, 4, 7, etc.). Thus, an integer represents a whole or complete number. But, it is often necessary to express quantities in terms of fractional parts of a whole number.

Decimal fractions are numbers whose positions have weights that are **negative powers of ten** such as  $10^{-1} = \frac{1}{10} = 0.1$ ,  $10^{-2} = \frac{1}{100} = 0.01$ , etc.

Figure 1-2 provides a condensed listing of negative powers of 10 (decimal fractions).

$$\begin{aligned}10^{-1} &= \frac{1}{10} = 0.1 \\10^{-2} &= \frac{1}{100} = 0.01 \\10^{-3} &= \frac{1}{1000} = 0.001 \\10^{-4} &= \frac{1}{10,000} = 0.0001 \\10^{-5} &= \frac{1}{100,000} = 0.00001 \\10^{-6} &= \frac{1}{1,000,000} = 0.000001\end{aligned}$$

Figure 1-2  
Condensed listing of negative  
powers of 10.

A radix point (decimal point for base 10 numbers) **separates** the **integer** and **fractional** parts of a number. The integer or whole portion is to the left of the decimal point and has positional weights of units, tens, hundreds, etc. The fractional part of the number is to the right of the decimal point and has positional weights of tenths, hundredths, thousandths, etc. To illustrate this, the decimal number 278.94 can be written with positional notation as shown below.

$$\begin{aligned}(2 \times 10^2) &+ (7 \times 10^1) + (8 \times 10^0) + (9 \times 10^{-1}) + (4 \times 10^{-2}) = \\(2 \times 100) &+ (7 \times 10) + (8 \times 1) + (9 \times 1/10) + (4 \times 1/100) = \\200 + 70 + 8 + 0.9 + 0.04 &= 278.94_{10}\end{aligned}$$

In this example, the left-most digit ( $2 \times 10^2$ ) is the **most significant digit** or MSD because it carries the greatest weight in determining the value of the number. The right-most digit, called the **least significant digit** or LSD, has the lowest weight in determining the value of the number. Therefore, as the term implies, the MSD is the digit that will affect the greatest change when its value is altered. The LSD has the smallest effect on the complete number value.

## Self-Test Review

1. The \_\_\_\_\_ indicates the number of characters or digits in a number system.
2. In the decimal number system, the base or radix is \_\_\_\_\_.
3. Write the following numbers using positional notation.
  - A.  $4563_{10}$
  - B.  $26.32_{10}$
  - C.  $536.9_{10}$
4. In the decimal number system, the radix point is called the \_\_\_\_\_.
5. Convert the following positional notations into their shorthand decimal form.
  - A.  $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2})$
  - B.  $(4 \times 10^{-1}) + (6 \times 10^{-2}) + (2 \times 10^{-3})$
  - C.  $(3 \times 10^3) + (7 \times 10^2) + (1 \times 10^1) + (0 \times 10^0)$
6. The radix point separates the \_\_\_\_\_ and \_\_\_\_\_ parts of a number.



## Answers

1. base or radix.
2. 10.
3. A.  $4563_{10} = 4000 + 500 + 60 + 3$ .  
 $= (4 \times 10^3) + (5 \times 10^2) + (6 \times 10^1) + (3 \times 10^0)$   
B.  $(2 \times 10^1) + (6 \times 10^0) + (3 \times 10^{-1}) + (2 \times 10^{-2})$   
C.  $(5 \times 10^2) + (3 \times 10^1) + (6 \times 10^0) + (9 \times 10^{-1})$
4. decimal point.
5. A.  $(5 \times 10^1) + (2 \times 10^0) + (3 \times 10^{-1}) + (8 \times 10^{-2}) =$   
 $(5 \times 10) + (2 \times 1) + (3 \times 1/10) + (8 \times 1/100) =$   
 $50 + 2 + 0.3 + 0.08 = 52.38_{10}$   
B.  $0.462_{10}$   
C.  $3710_{10}$
6. integer or whole, fractional.

## BINARY NUMBER SYSTEM

The simplest number system that uses positional notation is the binary number system. As the name implies, a **binary** system contains only two elements or states. In a number system this is expressed as a base of 2, using the digits 0 and 1. These two digits have the same basic value as 0 and 1 in the decimal number system.

Because of its simplicity, microprocessors use the binary number system to manipulate data. Binary data is represented by binary digits called **bits**. The term bit is derived from the contraction of **binary digit**. Microprocessors operate on groups of bits which are referred to as words. The binary number 11101101 contains eight bits.

### Positional Notation

As with the decimal number system, each bit (digit) position of a binary number carries a particular weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example 2). To evaluate the total quantity of a number, consider the specific bits and the weights of their positions. (Refer to Figure 1-3 for a condensed listing of powers of 2.) For example, the binary number 110101 can be written with positional notation as follows:

$$(1 \times 2^5) + (1 \times 2^4) + (0 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$$

To determine the decimal value of the binary number 110101, multiply each bit by its positional weight and add the results.

$$(1 \times 32) + (1 \times 16) + (0 \times 8) + (1 \times 4) + (0 \times 2) + (1 \times 1) = \\ 32 + 16 + 0 + 4 + 0 + 1 = 53_{10}$$

|                 |                      |
|-----------------|----------------------|
| $2^0 = 1_{10}$  | $2^6 = 64_{10}$      |
| $2^1 = 2_{10}$  | $2^7 = 128_{10}$     |
| $2^2 = 4_{10}$  | $2^8 = 256_{10}$     |
| $2^3 = 8_{10}$  | $2^9 = 512_{10}$     |
| $2^4 = 16_{10}$ | $2^{10} = 1024_{10}$ |
| $2^5 = 32_{10}$ | $2^{11} = 2048_{10}$ |

Figure 1-3

Condensed listing of powers of 2.

Fractional binary numbers are expressed as negative powers of 2. Figure 1-4 provides a condensed listing of negative powers of 2. In positional notation, the binary number 0.1101 can be expressed as follows:

$$(1 \times 2^{-1}) + (1 \times 2^{-2}) + (0 \times 2^{-3}) + (1 \times 2^{-4})$$

To determine the decimal value of the binary number 0.1101, multiply each bit by its positional weight and add the results.

$$(1 \times 1/2) + (1 \times 1/4) + (0 \times 1/8) + (1 \times 1/16) = \\ 0.5 + 0.25 + 0 + 0.0625 = 0.8125_{10}$$

In the binary number system, the radix point is called the binary point.

$$2^{-1} = \frac{1}{2} = 0.5_{10}$$

$$2^{-2} = \frac{1}{4} = 0.25_{10}$$

$$2^{-3} = \frac{1}{8} = 0.125_{10}$$

$$2^{-4} = \frac{1}{16} = 0.0625_{10}$$

$$2^{-5} = \frac{1}{32} = 0.03125_{10}$$

$$2^{-6} = \frac{1}{64} = 0.015625_{10}$$

$$2^{-7} = \frac{1}{128} = 0.0078125_{10}$$

$$2^{-8} = \frac{1}{256} = 0.00390625_{10}$$

Figure 1-4

Condensed listing of negative  
powers of 2.




## Converting Between the Binary and Decimal Number Systems

In working with microprocessors, you will often need to determine the decimal value of binary numbers. In addition, you will find it necessary to convert a specific decimal number into its binary equivalent. The following information shows how such conversions are accomplished.

**Binary to Decimal** To convert a binary number into its decimal equivalent, add together the weights of the positions in the number where binary 1's occur. The weights of the integer and fractional positions are indicated below.

| INTEGER |       |       |       |       |       |       |       | FRACTIONAL |          |          |
|---------|-------|-------|-------|-------|-------|-------|-------|------------|----------|----------|
| $2^7$   | $2^6$ | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | $2^{-1}$   | $2^{-2}$ | $2^{-3}$ |
| 128     | 64    | 32    | 16    | 8     | 4     | 2     | 1     | .5         | .25      | .125     |

Binary Point 

As an example, convert the binary number 1010 into its decimal equivalent. Since no binary point is shown, the number is assumed to be an integer number, where the binary point is to the right of the number. The right-most bit, called the **least significant bit** or LSB, has the lowest integer weight of  $2^0 = 1$ . The left-most bit is the **most significant bit** (MSB) because it carries the greatest weight in determining the value of the number. In this example, it has a weight of  $2^3 = 8$ . To evaluate the number, add together the weights of the positions where binary 1's appear. In this example, 1's occur in the  $2^3$  and  $2^1$  positions. The decimal equivalent is ten.

|                    |       |       |       |       |   |   |   |   |           |
|--------------------|-------|-------|-------|-------|---|---|---|---|-----------|
| Binary Number      | 1     | 0     | 1     | 0     |   |   |   |   |           |
| Position Weights   | $2^3$ | $2^2$ | $2^1$ | $2^0$ |   |   |   |   |           |
| Decimal Equivalent | 8     | +     | 0     | +     | 2 | + | 0 | = | $10_{10}$ |

To further illustrate this process, convert the binary number 101101.11 into its decimal equivalent.

|                    |       |       |       |       |       |       |          |          |   |   |   |   |    |   |     |   |              |
|--------------------|-------|-------|-------|-------|-------|-------|----------|----------|---|---|---|---|----|---|-----|---|--------------|
| Binary Number      | 1     | 0     | 1     | 1     | 0     | 1     | .1       | 1        |   |   |   |   |    |   |     |   |              |
| Position Weights   | $2^5$ | $2^4$ | $2^3$ | $2^2$ | $2^1$ | $2^0$ | $2^{-1}$ | $2^{-2}$ |   |   |   |   |    |   |     |   |              |
| Decimal Equivalent | 32    | +     | 0     | +     | 8     | +     | 4        | +        | 0 | + | 1 | + | .5 | + | .25 | = | $45.75_{10}$ |

**Decimal to Binary** A decimal integer number can be converted to a different base or radix through successive divisions by the desired base. To convert a decimal integer number to its binary equivalent, successively divide the number by 2 and note the remainders. When you divide by 2, the remainder will always be 1 or 0.

The remainders form the equivalent binary number.

As an example, the decimal number 25 is converted into its binary equivalent.

$$\begin{array}{rcl}
 25 \div 2 = 12 & \text{with remainder } 1 & \leftarrow \text{LSB} \\
 12 \div 2 = 6 & & 0 \\
 6 \div 2 = 3 & & 0 \\
 3 \div 2 = 1 & & 1 \\
 1 \div 2 = 0 & & 1 \leftarrow \text{MSB}
 \end{array}$$

Divide the decimal number by 2 and note the remainder. Then divide the quotient by 2 and again note the remainder. Then divide the quotient by 2 and again note the remainder. Continue this division process until 0 results. Then collect remainders beginning with the last or most significant bit (MSB) and proceed to the first or least significant bit (LSB). The number  $11001_2 = 25_{10}$ . Notice that the remainders are collected in the reverse order. That is, the first remainder becomes the least significant bit, while the last remainder becomes the most significant bit.

**NOTE:** Do not attempt to use a calculator to perform this conversion. It would only supply you with confusing results.

To further illustrate this, the decimal number 175 is converted into its binary equivalent.

$$\begin{array}{rcl}
 175 \div 2 = 87 & \text{with remainder } 1 & \leftarrow \text{LSB} \\
 87 \div 2 = 43 & & 1 \\
 43 \div 2 = 21 & & 1 \\
 21 \div 2 = 10 & & 1 \\
 10 \div 2 = 5 & & 0 \\
 5 \div 2 = 2 & & 1 \\
 2 \div 2 = 1 & & 0 \\
 1 \div 2 = 0 & & 1 \leftarrow \text{MSB}
 \end{array}$$

The division process continues until 0 results. The remainders are collected to produce the number  $10101111_2 = 175_{10}$ .

To convert a decimal fraction to a different base or radix, multiply the fraction successively by the desired base and record any integers produced by the multiplication as an overflow. For example, to convert the decimal fraction 0.3125 into its binary equivalent, multiply repeatedly by 2.

|                                   |               |   |       |
|-----------------------------------|---------------|---|-------|
| $0.3125 \times 2 = 0.625 = 0.625$ | with overflow | 0 | ← MSB |
| $0.6250 \times 2 = 1.250 = 0.250$ |               | 1 |       |
| $0.2500 \times 2 = 0.500 = 0.500$ |               | 0 |       |
| $0.5000 \times 2 = 1.000 = 0$     |               | 1 | ← LSB |

These multiplications will result in numbers with a 1 or 0 in the units position (the position to the left of the decimal point). By recording the value of the units position, you can construct the equivalent binary fraction. This units position value is called the "overflow." Therefore, when 0.3125 is multiplied by 2, the overflow is 0. This becomes the most significant bit (MSB) of the binary equivalent fraction. Then 0.625 is multiplied by 2. Since the product is 1.25, the overflow is 1. When there is an overflow of 1, it is effectively subtracted from the product when the value is recorded. Therefore, only 0.25 is multiplied by 2 in the next multiplication process. This method continues until an overflow with no fraction results. It is important to note that you can not always obtain 0 when you multiply by 2. Therefore, you should only continue the conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (binary) point with the MSB and proceed to the LSB. This is the same order in which the overflows were produced. The number  $0.0101_2 = 0.3125_{10}$ .

To further illustrate this process, the decimal fraction 0.84375 is converted into its binary equivalent.

|                                      |               |   |       |
|--------------------------------------|---------------|---|-------|
| $0.90625 \times 2 = 1.8125 = 0.8125$ | with overflow | 1 | ← MSB |
| $0.81250 \times 2 = 1.6250 = 0.6250$ |               | 1 |       |
| $0.62500 \times 2 = 1.2500 = 0.2500$ |               | 1 |       |
| $0.25000 \times 2 = 0.5000 = 0.5000$ |               | 0 |       |
| $0.50000 \times 2 = 1.0000 = 0$      |               | 1 | ← LSB |

The multiplication process continues until either 0 or the desired precision is obtained. The overflows are then collected beginning with the MSB at the binary (radix) point and proceeding to the LSB. The number  $0.11101_2 = 0.90625_{10}$ .



If the decimal number contains both an integer and fraction, you must separate the integer and fraction using the decimal point as the break point. Then perform the appropriate conversion process on each number portion. After you convert the binary integer and binary fraction, recombine them. For example, the decimal number 14.375 is converted into its binary equivalent.

$$14.375_{10} = 14_{10} + 0.375_{10}$$

$$14 \div 2 = 7$$

with remainder 0 ← LSB

$$7 \div 2 = 3$$

1

$$3 \div 2 = 1$$

1

$$1 \div 2 = 0$$

1

← MSB

$$14_{10} = 1110_2$$

$$0.375 \times 2 = 0.75 = 0.75$$

with overflow 0 ← MSB

$$0.750 \times 2 = 1.50 = 0.50$$

1

$$0.500 \times 2 = 1.00 = 0$$

1

← LSB

$$0.375_{10} = 0.011_2$$

$$14.375_{10} = 14_{10} + 0.375_{10} = 1110_2 + 0.011_2 = 1110.011_2.$$

## Self-Test Review

7. The base or radix of the binary number system is \_\_\_\_\_.
8. A binary digit is called a \_\_\_\_\_.
9. Convert the following binary integers to decimal.
  - A. 101101
  - B. 1001
  - C. 1101100
10. Convert the following binary fractions to decimal.
  - A. 0.011
  - B. 0.01101
  - C. 0.1001
11. Convert the following decimal integers to binary.
  - A. 63
  - B. 12
  - C. 132
12. Convert the following decimal fractions to binary.
  - A. 0.4375
  - B. 0.96875
  - C. 0.625
13. Convert  $13.125_{10}$  to binary.

## Answers

7. 2
8. bit
9. A.  $101101_2 = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 32 + 0 + 8 + 4 + 0 + 1 = 45_{10}$   
 B.  $1001_2 = 9_{10}$   
 C.  $1101100_2 = 108_{10}$
10. A.  $0.011_2 = (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3}) = 0 + \frac{1}{4} + \frac{1}{8} = 0 + 0.25 + 0.125 = 0.375_{10}$   
 B.  $0.01101_2 = 0.40625_{10}$   
 C.  $0.1001_2 = 0.5625_{10}$
11. A.  $63 \div 2 = 31$  with remainder 1 ← LSB  
 $31 \div 2 = 15$  1  
 $15 \div 2 = 7$  1  
 $7 \div 2 = 3$  1  
 $3 \div 2 = 1$  1  
 $1 \div 2 = 0$  1 ← MSB  
 $63_{10} = 111111_2$   
 B.  $12_{10} = 1100_2$   
 C.  $132_{10} = 10000100_2$



12. A.  $0.4375 \times 2 = 0.875 = 0.875$  with overflow 0  $\leftarrow$  MSB  
 $0.8750 \times 2 = 1.750 = 0.750$  1  
 $0.7500 \times 2 = 1.500 = 0.500$  1  
 $0.5000 \times 2 = 1.000 = 0$  1  $\leftarrow$  LSB

$0.4375_{10} = 0.0111_2$

B.  $0.96875_{10} = 0.11111_2$

C.  $0.625_{10} = 0.101_2$

13.  $13.125_{10} = 13_{10} + 0.125_{10}$

$13 \div 2 = 6$  with remainder 1  $\leftarrow$  LSB  
 $6 \div 2 = 3$  0  
 $3 \div 2 = 1$  1  
 $1 \div 2 = 0$  1  $\leftarrow$  MSB

$13_{10} = 1101_2$

$0.125 \times 2 = 0.25 = 0.25$  with overflow 0  $\leftarrow$  MSB  
 $0.250 \times 2 = 0.50 = 0.50$  0  
 $0.500 \times 2 = 1.00 = 0$  1  $\leftarrow$  LSB

$0.125_{10} = 0.001_2$

$13.125_{10} = 13_{10} + 0.125_{10} = 1101_2 + 0.001_2 = 1101.001_2$

## OCTAL NUMBER SYSTEM

Octal is another number system that is often used with microprocessors. It has a base (radix) of 8, and uses the digits 0 through 7. These eight digits have the same basic value as the digits 0—7 in the decimal number system.

As with the binary number system, each digit position of an octal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example, 8). To evaluate the total quantity of a number, consider the specific digits and the weights of their positions. Refer to Figure 1-5 for a condensed listing of powers of 8. For example, the octal number 372.01 can be written with positional notation as follows:

$$(3 \times 8^2) + (7 \times 8^1) + (2 \times 8^0) + (0 \times 8^{-1}) + (1 \times 8^{-2})$$

The decimal value of the octal number 372.01 is determined by multiplying each digit by its positional weight and adding the results. As with decimal and binary numbers, the radix (octal) point separates the integer from the fractional part of the number.

$$(3 \times 64) + (7 \times 8) + (2 \times 1) + (0 \times 0.125) + (1 \times 0.015625) = 192 + 56 + 2 + 0 + 0.015625 = 250.015625_{10}$$

$$8^{-4} = \frac{1}{4096} = 0.000244140625_{10}$$

$$8^{-3} = \frac{1}{512} = 0.001953125_{10}$$

$$8^{-2} = \frac{1}{64} = 0.015625_{10}$$

$$8^{-1} = \frac{1}{8} = 0.125_{10}$$

$$1_{10} = 8^0$$

$$8_{10} = 8^1$$

$$64_{10} = 8^2$$

$$512_{10} = 8^3$$

$$4096_{10} = 8^4$$

$$32768_{10} = 8^5$$

$$262144_{10} = 8^6$$

Figure 1-5

Condensed listing of powers of 8.

## Conversion From Decimal to Octal

Decimal to octal conversion is accomplished in the same manner as decimal to binary, with one exception; the base number is now 8 rather than 2. As an example, the decimal number 194 is converted into its octal equivalent.

$$194 \div 8 = 24 \text{ with remainder } 2 \quad \leftarrow \text{LSD}$$

$$24 \div 8 = 3 \quad 0$$

$$3 \div 8 = 0 \quad 3 \quad \leftarrow \text{MSD}$$

Divide the decimal by 8 and note the remainder. (The remainder can be any number from 0 to 7.)

Then divide the quotient by 8 and again note the remainder. Continue dividing until 0 results. Finally, collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number  $302_8 = 194_{10}$ . Figure 1-6 illustrates the relationship between the first several decimal, octal, and binary integers.

| DECIMAL | OCTAL | BINARY |
|---------|-------|--------|
| 0       | 0     | 0      |
| 1       | 1     | 1      |
| 2       | 2     | 10     |
| 3       | 3     | 11     |
| 4       | 4     | 100    |
| 5       | 5     | 101    |
| 6       | 6     | 110    |
| 7       | 7     | 111    |
| 8       | 10    | 1000   |
| 9       | 11    | 1001   |
| 10      | 12    | 1010   |
| 11      | 13    | 1011   |
| 12      | 14    | 1100   |
| 13      | 15    | 1101   |
| 14      | 16    | 1110   |
| 15      | 17    | 1111   |
| 16      | 20    | 10000  |
| 17      | 21    | 10001  |
| 18      | 22    | 10010  |
| 19      | 23    | 10011  |
| 20      | 24    | 10100  |

Figure 1-6

Sample comparison of decimal, octal, and binary integers.



To further illustrate this process, the decimal number 175 is converted into its octal equivalent.

$$\begin{array}{rcl} 175 \div 8 = 21 \text{ with remainder } 7 & \leftarrow & \text{LSD} \\ 21 \div 8 = 2 & & 5 \\ 2 \div 8 = 0 & & 2 \leftarrow \text{MSD} \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number  $257_8 = 175_{10}$ .

To convert a decimal fraction to an octal fraction, multiply the fraction successively by 8 (octal base). As an example, the decimal fraction 0.46875 is converted into its octal equivalent.

$$\begin{array}{rcl} 0.46875 \times 8 = 3.75 = 0.75 \text{ with overflow } 3 & \leftarrow & \text{MSD} \\ 0.75000 \times 8 = 6.00 = 0 & & 6 \leftarrow \text{LSD} \end{array}$$

Multiply the decimal number by 8. If the product exceeds one, subtract the integer (overflow) from the product. Then multiply the product fraction by 8 and again note any "overflow." Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 8. Therefore, you should only continue this conversion process to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix (octal point) with the MSD and proceed to the LSD. The number  $0.36_8 = 0.46875_{10}$ . Figure 1-7 illustrates the relationship between decimal, octal, and binary fractions.

Now, the decimal fraction 0.136 will be converted into its octal equivalent with four-place precision.

$$\begin{array}{rcl} 0.136 \times 8 = 1.088 = 0.088 \text{ with overflow } 1 & \leftarrow & \text{MSD} \\ 0.088 \times 8 = 0.704 = 0.704 & & 0 \\ 0.704 \times 8 = 5.632 = 0.632 & & 5 \\ 0.632 \times 8 = 5.056 = 0.056 & & 5 \leftarrow \text{LSD} \\ 0.136_{10} \approx 0.1055_8 \end{array}$$

The number  $0.1055_8$  approximately equals  $0.136_{10}$ . If you convert  $0.1055_8$  back to decimal (using positional notation), you will find  $0.1055_8 = 0.135986328125_{10}$ . This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

| DECIMAL  | OCTAL | BINARY   |
|----------|-------|----------|
| 0.015625 | 0.01  | 0.000001 |
| 0.03125  | 0.02  | 0.00001  |
| 0.046875 | 0.03  | 0.000011 |
| 0.0625   | 0.04  | 0.0001   |
| 0.078125 | 0.05  | 0.000101 |
| 0.09375  | 0.06  | 0.00011  |
| 0.109375 | 0.07  | 0.000111 |
| 0.125    | 0.1   | 0.001    |
| 0.140625 | 0.11  | 0.001001 |
| 0.15625  | 0.12  | 0.00101  |
| 0.171875 | 0.13  | 0.001011 |
| 0.1875   | 0.14  | 0.0011   |
| 0.203125 | 0.15  | 0.001101 |
| 0.21875  | 0.16  | 0.00111  |
| 0.234375 | 0.17  | 0.001111 |
| 0.25     | 0.2   | 0.01     |
| 0.265625 | 0.21  | 0.010001 |
| 0.28125  | 0.22  | 0.01001  |
| 0.296875 | 0.23  | 0.010011 |
| 0.3125   | 0.24  | 0.0101   |

Figure 1-7  
Sample comparison of decimal,  
octal, and binary fractions.

As with decimal to binary conversion of a number that contains both an integer and fraction, decimal to octal conversion requires two operations. You must separate the integer from the fraction, then perform the appropriate conversion on each number. After you convert them, you must recombine the octal integer and octal fraction. For example, convert the decimal number 124.78125 into its octal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 8 = 15 \quad \text{with remainder} \quad 4 \quad \leftarrow \text{LSD}$$

$$15 \div 8 = 1 \quad 7$$

$$1 \div 8 = 0 \quad 1 \quad \leftarrow \text{MSD}$$

$$124_{10} = 174_8$$

$$0.78125 \times 8 = 6.25 = 0.25 \quad \text{with overflow} \quad 6 \quad \leftarrow \text{MSD}$$

$$0.25000 \times 8 = 2.00 = 0 \quad 2 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.62_8$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 174_8 + 0.62_8 = 174.62_8$$

## Converting Between the Octal and Binary Number Systems

Microprocessors manipulate data using the binary number system. However, when larger quantities are involved, the binary number system can become cumbersome. Therefore, other number systems are frequently used as a form of binary shorthand to speed-up and simplify data entry and display. The octal number system is one of the systems that is used in this manner. It is similar to the decimal number system, which makes it easier to understand numerical values. In addition, conversion between binary and octal is readily accomplished because of the value structure of octal. Figures 1-6 and 1-7 illustrate the relationship between octal and binary integers and fractions.

As you know, three bits of a binary number exactly equal eight value combinations. Therefore, you can represent a 3-bit binary number with a 1-digit octal number.

$$101_2 = (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 4 + 0 + 1 = 5_8$$

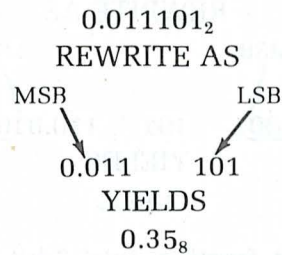
Because of this relationship, converting binary to octal is simple and straight forward. For example, binary number 101001 is converted into its octal equivalent.

101001<sub>2</sub>  
REWRITE AS  
 MSB                      LSB  
 ↙                      ↘  
 101                      001  
 YIELDS  
 51<sub>8</sub>

To convert a binary number to octal, first separate the number into groups containing three bits, beginning with the least significant bit. Then convert each 3-bit group into its octal equivalent. This gives you an octal number equal in value to the binary number.

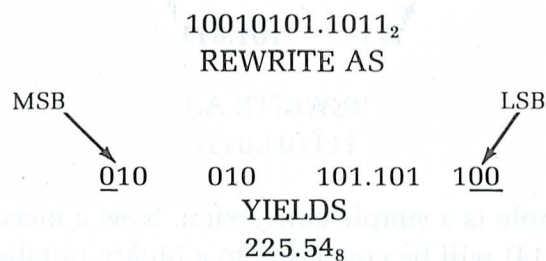


Binary fractions can also be converted to their octal equivalents using the same process, with one exception. The binary bits must be separated into groups of three beginning with the most significant bit. For example, the binary fraction  $0.011101_2$  is converted into its octal equivalent.



Again, you must first separate the binary number into groups of three beginning at the radix (binary) point. Then convert each 3-bit group into its octal equivalent.

To separate binary numbers into 3-bit groups when the number does not contain the necessary bits, add zeros to the number until the number can be separated into 3-bit groups. For example, binary number  $10010101.1011_2$  is converted into its octal equivalent.



As before, the integer part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the third group contains only two bits. However, a zero can be added to the group without changing the value of the binary number. Next, the fractional part of the number is separated into 3-bit groups, beginning at the radix (binary) point. Note that the second group contains only one bit. By adding two zeros to the group, the group is complete with no change in the value of the binary number.

**NOTE:** Whenever you add zeros to a **binary integer**, always place them to the **left** of the most significant bit. When you add zeros to a **binary fraction**, always place them to the **right** of the least significant bit.

After you have formed the 3-bit groups, convert each group into its octal equivalent. This gives you an octal number equal in value to the binary number. Now convert binary number 1101110.01 into its octal equivalent.

1101110.01<sub>2</sub>  
REWRITE AS

|          |     |     |         |          |
|----------|-----|-----|---------|----------|
| MSB<br>↓ | 001 | 101 | 110.010 | LSB<br>↓ |
|----------|-----|-----|---------|----------|

YIELDS  
156.2<sub>8</sub>

Separate the integer and fraction into 3-bit groups, adding zeros as necessary. Then convert each 3-bit group to octal. **Never** shift the radix (binary) point in order to form 3-bit groups.

Converting octal to binary is just the opposite of the previous process. You simply convert each octal number into its 3-bit binary equivalent. For example, convert the octal number 75.3 into its binary equivalent.

75.3<sub>8</sub>  
YIELDS

|          |     |         |          |
|----------|-----|---------|----------|
| MSB<br>↓ | 111 | 101.011 | LSB<br>↓ |
|----------|-----|---------|----------|

REWRITE AS  
111101.011<sub>2</sub>

The above example is a simple conversion. Now a more complex octal number (1752.714) will be converted to a binary number.

1752.714<sub>8</sub>  
YIELDS

|          |     |     |     |         |     |     |          |
|----------|-----|-----|-----|---------|-----|-----|----------|
| MSB<br>↓ | 001 | 111 | 101 | 010.111 | 001 | 100 | LSB<br>↓ |
|----------|-----|-----|-----|---------|-----|-----|----------|

REWRITE AS  
1111101010.1110011<sub>2</sub>

Again, each octal digit is converted into its 3-bit binary equivalent. However, in this example, there are two insignificant zeros in front of the MSB and after the LSB. Since these zeros have no value, they should be removed from the final result.

## Self-Test Review

14. The base or radix of the octal number system is \_\_\_\_\_.
15. Convert the following decimal integers to octal.
- A. 156
  - B. 32
  - C. 1785
16. Convert the following decimal fractions to octal. Do not use greater than 4-place precision.
- A. 0.1432
  - B. 0.8125
  - C. 0.6832
17. Convert  $735.984375_{10}$  to octal.
18. Convert the following binary numbers to octal.
- A. 10000111.01101
  - B. 11101.0101
  - C. 1001101.000001
19. Convert the following octal numbers to binary.
- A. 372.61
  - B. 11.001
  - C. 3251.034



## Answers

14. 8

15. A.  $156 \div 8 = 19$  with remainder 4  $\leftarrow$  LSD  
 $19 \div 8 = 2$  3  
 $2 \div 8 = 0$  2  $\leftarrow$  MSD

$$156_{10} = 234_8$$

B.  $32_{10} = 40_8$

C.  $1785_{10} = 3371_8$

16. A.  $0.1432 \times 8 = 1.1456 = 0.1456$  overflow 1  $\leftarrow$  MSD  
 $0.1456 \times 8 = 1.1648 = 0.1648$  1  
 $0.1648 \times 8 = 1.3184 = 0.3184$  1  
 $0.3184 \times 8 = 2.5472 = 0.5472$  2  $\leftarrow$  LSD

$$0.1432_{10} = 0.1112_8$$

B.  $0.8125_{10} = 0.64_8$

C.  $0.6832_{10} = 0.5356_8$

17.  $735.984375_{10} = 735_{10} + 0.984375_{10}$   
 $= 735 \div 8 = 91$  with remainder 7  $\leftarrow$  LSD  
 $91 \div 8 = 11$  3  
 $11 \div 8 = 1$  3  
 $1 \div 8 = 0$  1  $\leftarrow$  MSD

$$735_{10} = 1337_8$$

$0.984375 \times 8 = 7.875 = 0.875$  overflow  
 7  $\leftarrow$  MSD

$0.875000 \times 8 = 7.00 = 0$  7  $\leftarrow$  LSD

$$0.984375_{10} = 0.77_8$$

$$735.984375_{10} = 735_{10} + 0.984375_{10} = 1337_8 + 0.77_8 = 1337.77_8$$

18. A.  $10000111.01101_2 = 010\ 000\ 111.011\ 010_2$   
 $= 207.32_8$

B.  $11101.0101_2 = 35.24_8$

C.  $1001101.000001_2 = 115.01_8$

19. A.  $372.61_8 = 011\ 111\ 010.110\ 001_2 = 11111010.110001_2$   
 B.  $11.001_8 = 1001.000000001_2$   
 C.  $3251.034_8 = 11010101001.0000111_2$

## HEXADECIMAL NUMBER SYSTEM

Hexadecimal is another number system that is often used with microprocessors. It is similar in value structure to the octal number system, and thus allows easy conversion with the binary number system. Because of this feature and the fact that hexadecimal simplifies data entry and display to a greater degree than octal, you will use hexadecimal more often than any other number system in this course. As the name implies, hexadecimal has a base (radix) of  $16_{10}$ . It uses the digits 0 through 9 and the letters A through F.

The letters are used because it is necessary to represent  $16_{10}$  different values with a single digit for each value. Therefore, the letters A through F are used to represent the number values  $10_{10}$  through  $15_{10}$ . The following discussion will compare the decimal number system with the hexadecimal number system.

All of the numbers are of equal value between systems ( $0_{10} = 0_{16}$ ,  $3_{10} = 3_{16}$ ,  $9_{10} = 9_{16}$ , etc.). For numbers greater than 9, this relationship exists:  $10_{10} = A_{16}$ ,  $11_{10} = B_{16}$ ,  $12_{10} = C_{16}$ ,  $13_{10} = D_{16}$ ,  $14_{10} = E_{16}$ , and  $15_{10} = F_{16}$ . Using letters in counting may appear awkward until you become familiar with the system. Figure 1-8 illustrates the relationship between decimal and hexadecimal integers, while Figure 1-9 illustrates the relationship between decimal and hexadecimal fractions.

| DECIMAL | HEXADECIMAL | BINARY |
|---------|-------------|--------|
| 0       | 0           | 0      |
| 1       | 1           | 1      |
| 2       | 2           | 10     |
| 3       | 3           | 11     |
| 4       | 4           | 100    |
| 5       | 5           | 101    |
| 6       | 6           | 110    |
| 7       | 7           | 111    |
| 8       | 8           | 1000   |
| 9       | 9           | 1001   |
| 10      | A           | 1010   |
| 11      | B           | 1011   |
| 12      | C           | 1100   |
| 13      | D           | 1101   |
| 14      | E           | 1110   |
| 15      | F           | 1111   |
| 16      | 10          | 10000  |
| 17      | 11          | 10001  |
| 18      | 12          | 10010  |
| 19      | 13          | 10011  |
| 20      | 14          | 10100  |
| 21      | 15          | 10101  |
| 22      | 16          | 10110  |
| 23      | 17          | 10111  |
| 24      | 18          | 11000  |
| 25      | 19          | 11001  |
| 26      | 1A          | 11010  |
| 27      | 1B          | 11011  |
| 28      | 1C          | 11100  |
| 29      | 1D          | 11101  |
| 30      | 1E          | 11110  |
| 31      | 1F          | 11111  |
| 32      | 20          | 100000 |
| 33      | 21          | 100001 |
| 34      | 22          | 100010 |
| 35      | 23          | 100011 |

Figure 1-8  
Sample comparison of decimal,  
hexadecimal, and binary integers.



| DECIMAL    | HEXADECIMAL | BINARY     |
|------------|-------------|------------|
| 0.00390625 | 0.01        | 0.00000001 |
| 0.0078125  | 0.02        | 0.0000001  |
| 0.01171875 | 0.03        | 0.00000011 |
| 0.015625   | 0.04        | 0.000001   |
| 0.01953125 | 0.05        | 0.00000101 |
| 0.0234375  | 0.06        | 0.0000011  |
| 0.02734375 | 0.07        | 0.00000111 |
| 0.03125    | 0.08        | 0.00001    |
| 0.03515625 | 0.09        | 0.00001001 |
| 0.0390625  | 0.0A        | 0.0000101  |
| 0.04296875 | 0.0B        | 0.00001011 |
| 0.046875   | 0.0C        | 0.00011    |
| 0.05078125 | 0.0D        | 0.00001101 |
| 0.0546875  | 0.0E        | 0.0000111  |
| 0.05859375 | 0.0F        | 0.00001111 |
| 0.0625     | 0.1         | 0.0001     |
| 0.06640625 | 0.11        | 0.00010001 |
| 0.0703125  | 0.12        | 0.0001001  |
| 0.07421875 | 0.13        | 0.00010011 |
| 0.078125   | 0.14        | 0.000101   |
| 0.08203125 | 0.15        | 0.00010101 |
| 0.0859375  | 0.16        | 0.0001011  |
| 0.08984375 | 0.17        | 0.00010111 |
| 0.09375    | 0.18        | 0.00011    |
| 0.09765625 | 0.19        | 0.00011001 |
| 0.1015625  | 0.1A        | 0.0001101  |
| 0.10546875 | 0.1B        | 0.00011011 |
| 0.109375   | 0.1C        | 0.000111   |
| 0.11328125 | 0.1D        | 0.00011101 |
| 0.1171875  | 0.1E        | 0.0001111  |
| 0.12109375 | 0.1F        | 0.00011111 |
| 0.125      | 0.2         | 0.001      |

Figure 1-9  
 Sample comparison of decimal,  
 hexadecimal, and binary fractions.

As with the previous number systems, each digit position of a hexadecimal number carries a positional weight which determines the magnitude of that number. The weight of each position is determined by some power of the number system base (in this example,  $16_{10}$ ). The total quantity of a number can be evaluated by considering the specific digits and the weights of their positions. (Refer to Figure 1-10 for a condensed listing of powers of  $16_{10}$ .) For example, the hexadecimal number E5D7.A3 can be written with positional notation as follows:

$$(E \times 16^3) + (5 \times 16^2) + (D \times 16^1) + (7 \times 16^0) + (A \times 16^{-1}) + (3 \times 16^{-2})$$

The decimal value of the hexadecimal number E5D7.A3 is determined by multiplying each digit by its positional weight and adding the results. As with the previous number systems, the radix (hexadecimal) point separates the integer from the fractional part of the number.

$$(14 \times 4096) + (5 \times 256) + (13 \times 16) + (7 \times 1) + (10 \times 1/16) + (3 \times 1/256) = \\ 57344 + 1280 + 208 + 7 + 0.625 + 0.01171875 = \\ 58839.63671875_{10}$$

$$\begin{aligned} 16^{-4} &= \frac{1}{65536} = 0.0000152587890625_{10} \\ 16^{-3} &= \frac{1}{4096} = 0.000244140625_{10} \\ 16^{-2} &= \frac{1}{256} = 0.00390625_{10} \\ 16^{-1} &= \frac{1}{16} = 0.0625_{10} \end{aligned}$$

$$\begin{aligned} 1_{10} &= 16^0 \\ 16_{10} &= 16^1 \\ 256_{10} &= 16^2 \\ 4096_{10} &= 16^3 \\ 65536_{10} &= 16^4 \\ 1048576_{10} &= 16^5 \\ 16777216_{10} &= 16^6 \end{aligned}$$

Figure 1-10  
Condensed listing of powers of 16.

## Conversion From Decimal to Hexadecimal

Decimal to hexadecimal conversion is accomplished in the same manner as decimal to binary or octal, but with a base number of  $16_{10}$ . As an example, the decimal number 156 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 156 \div 16 = 9 & \text{with remainder } 12 = C & \leftarrow & \text{LSD} \\ 9 \div 16 = 0 & & 9 = 9 & \leftarrow \text{MSD} \end{array}$$

Divide the decimal number by  $16_{10}$  and note the remainder. If the remainder exceeds 9, convert the 2-digit number to its hexadecimal equivalent ( $12_{10} = C$  in this example). Then divide the quotient by 16 and again note the remainder. Continue dividing until a quotient of 0 results. Then collect the remainders beginning with the last or most significant digit (MSD) and proceed to the first or least significant digit (LSD). The number  $9C_{16} = 156_{10}$ . NOTE: The letter H after a number is sometimes used to indicate hexadecimal. However, this course will always use the subscript 16.

To further illustrate this, the decimal number 47632 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 47632 \div 16 = 2977 & \text{with remainder } 0 = 0 & \leftarrow & \text{LSD} \\ 2977 \div 16 = 186 & & 1 = 1 & \\ 186 \div 16 = 11 & & 10 = A & \\ 11 \div 16 = 0 & & 11 = B & \leftarrow \text{MSD} \end{array}$$

The division process continues until a quotient of 0 results. The remainders are collected, producing the number  $BA10_{16} = 47632_{10}$ . Remember, any remainder that exceeds the digit 9 must be converted to its letter equivalent. (In this example,  $10 = A$ , and  $11 = B$ .)

To convert a decimal fraction to a hexadecimal fraction, multiply the fraction successively by  $16_{10}$  (hexadecimal base). As an example the decimal fraction 0.78125 is converted into its hexadecimal equivalent.

$$\begin{array}{rcll} 0.78125 \times 16 = 12.5 = 0.5 & \text{with overflow } 12 = C & \leftarrow & \text{MSD} \\ 0.50000 \times 16 = 8.0 = 0 & & 8 = 8 & \leftarrow \text{LSD} \end{array}$$



Multiply the decimal by  $16_{10}$ . If the product exceeds one, subtract the integer (overflow) from the product. If the "overflow" exceeds 9, convert the 2-digit number to its hexadecimal equivalent. Then multiply the product fraction by  $16_{10}$  and again note any overflow. Continue multiplying until an overflow, with 0 for a fraction, results. Remember, you can not always obtain 0 when you multiply by 16. Therefore, you should only continue the conversion to the accuracy or precision you desire. Collect the conversion overflows beginning at the radix point with the MSD and proceed to the LSD. The number  $0.C8_{16} = 0.78125_{10}$ .

Now the decimal fraction 0.136 will be converted into its hexadecimal equivalent with five-place precision.

|                            |           |          |        |       |
|----------------------------|-----------|----------|--------|-------|
| $0.136 \times 16 = 2.176$  | $= 0.176$ | overflow | 2 = 2  | → MSD |
| $0.176 \times 16 = 2.816$  | $= 0.816$ |          | 2 = 2  |       |
| $0.816 \times 16 = 13.056$ | $= 0.056$ |          | 13 = D |       |
| $0.056 \times 16 = 0.896$  | $= 0.896$ |          | 0 = 0  |       |
| $0.896 \times 16 = 14.336$ | $= 0.336$ |          | 14 = E | → LSD |

The number  $0.22D0E_{16}$  approximately equals  $0.136_{10}$ . If you convert  $0.22D0E_{16}$  back to decimal (using positional notation), you will find  $0.22D0E_{16} = 0.1359996795654296875_{10}$ . This example shows that extending the precision of your conversion is of little value unless extreme accuracy is required.

As shown in this section, conversion of an integer from decimal to hexadecimal requires a different technique than for conversion of a fraction. Therefore, when you convert a hexadecimal number composed of an integer and a fraction, you must separate the integer and fraction, then perform the appropriate operation on each. After you convert them, you must recombine the integer and fraction. For example, the decimal number 124.78125 is converted into its hexadecimal equivalent.

$$124.78125_{10} = 124_{10} + 0.78125_{10}$$

$$124 \div 16 = 7 \quad \text{with remainder } 12 = C \quad \leftarrow \text{LSD}$$

$$7 \div 16 = 0 \quad \quad \quad 7 = 7 \quad \leftarrow \text{MSD}$$

$$124_{10} = 7C_{16}$$

$$0.78125 \times 16 = 12.5 = 0.5 \quad \text{overflow} \quad 12 = C \quad \leftarrow \text{MSD}$$

$$0.50000 \times 16 = 8.0 = 0 \quad \quad \quad 8 = 8 \quad \leftarrow \text{LSD}$$

$$0.78125_{10} = 0.C8_{16}$$

$$124.78125_{10} = 124_{10} + 0.78125_{10} = 7C_{16} + 0.C8_{16} = 7C.C8_{16}$$

First separate the decimal integer and fraction. Then convert the integer and fraction to hexadecimal.

Finally, recombine the integer and fraction.

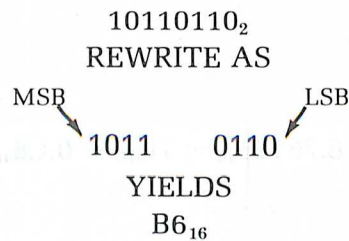
## Converting Between the Hexadecimal and Binary Number Systems

Previously, the octal number system was described as an excellent short-hand form to express large binary quantities. This method is very useful with many microprocessors. The trainer used with this course uses the hexadecimal number system to represent binary quantities. As a result, frequent conversions from binary-to-hexadecimal are necessary. Figures 1-8 and 1-9 illustrate the relationship between hexadecimal and binary integers and fractions.

As you know, four bits of a binary number exactly equal  $16_{10}$  value combinations. Therefore, you can represent a 4-bit binary number with a 1-digit hexadecimal number:

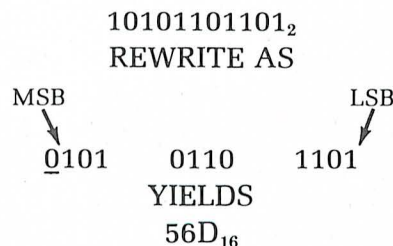
$$1101_2 = (1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) = 8 + 4 + 0 + 1 = 13_{10} = D_{16}$$

Because of this relationship, converting binary to hexadecimal is simple and straightforward. For example, binary number 10110110 is converted into its hexadecimal equivalent.



To convert a binary number to hexadecimal, first separate the number into groups containing four bits, beginning with the least significant bit. Then convert each 4-bit group into its hexadecimal equivalent. Don't forget to use letter digits as required. This gives you a hexadecimal number equal in value to the binary number.

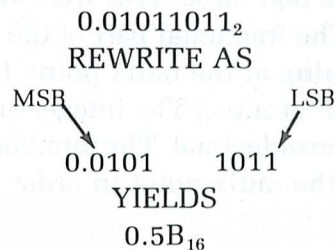
Now convert a larger binary number (10101101101) into its hexadecimal equivalent.





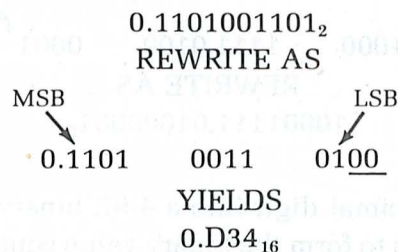
Again, the binary number is separated into 4-bit groups beginning with the LSB. However, the third group contains only three bits. Since each group must contain four bits, a zero must be added after the MSB. The third group will then have four bits with no change in the value of the binary number. Now each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary integer, always place them to the left of the most significant bit.**

Binary fractions can also be converted to their hexadecimal equivalents using the same process, with one exception; the binary bits are separated into groups of four, beginning with the most significant bit (at the radix point). For example, the binary fraction 0.01011011 is converted into its hexadecimal equivalent.



Again, you must separate the binary number into groups of four, beginning with the radix point. Then convert each 4-bit group into its hexadecimal equivalent. This gives you a hexadecimal number equal in value to the binary number.

Now convert a larger binary fraction (0.1101001101) into its hexadecimal equivalent.



Separate the binary number into 4-bit groups, beginning at the radix (binary) point (MSB). Note that the third group contains only two bits. Since each group must contain four bits, two zeros must be added after the LSB. The third group will then have four bits with no change in the value of the binary number. Now, each 4-bit group can be converted into its hexadecimal equivalent. **Whenever you add zeros to a binary fraction, always place them to the right of the least significant bit.**

Now, a binary number containing both an integer and a fraction ( $110110101.01110111_2$ ) will be converted into its hexadecimal equivalent.

$110110101.01110111_2$   
REWRITE AS

MSB ↙
0001
1011
0101.0111
0111 ↘
LSB

YIELDS  
 $1B5.77_{16}$

The integer part of the number is separated into groups of four, **beginning** at the radix point. Note that three zeros were added to the third group to complete the group. The fractional part of the number is separated into groups of four, **beginning** at the radix point. (No zeros were needed to complete the fractional groups.) The integer and fractional 4-bit groups are then converted to hexadecimal. The number  $110110101.01110111_2 = 1B5.77_{16}$ . **Never** shift the radix point in order to form 4-bit groups.

Converting hexadecimal to binary is just the opposite of the previous process; simply convert each hexadecimal number into its 4-bit binary equivalent. For example, convert the hexadecimal number  $8F.41_{16}$  into its binary equivalent.

$8F.41_{16}$   
YIELDS

MSB ↙
1000
1111.0100
0001 ↘
LSB

REWRITE AS  
 $10001111.01000001_2$

Convert each hexadecimal digit into a 4-bit binary number. Then condense the 4-bit groups to form the binary value equal to the hexadecimal value. The number  $8F.41_{16} = 10001111.01000001_2$ .

Now, the hexadecimal number  $175.4E_{16}$  will be converted into its binary equivalent.

$175.4E_{16}$   
YIELDS  
MSB                      LSB  
↓                              ↓  
0001 0111 0101.0100 1110  
REWRITE AS  
 $101110101.0100111_2$

Again, each hexadecimal digit is converted into its 4-bit binary equivalent. However, in this example there are three insignificant zeros in front of the MSB and one after the LSB. Since these zeros have no value, they should be removed from the final result.



## Self-Test Review

20. The base or radix of the hexadecimal number system is \_\_\_\_\_.
21. Convert the following decimal integers to hexadecimal.
- A. 783
  - B. 5372
  - C. 957
22. Convert the following decimal fractions to hexadecimal. Do not use greater than four-place precision.
- A. 0.653
  - B. 0.109375
  - C. 0.4567
23. Convert  $1573.125_{10}$  to its hexadecimal equivalent.
24. Convert the following binary numbers to hexadecimal.
- A. 100001101.01011
  - B. 11111011001.01
  - C. 110001101.00010010101
25. Convert the following hexadecimal numbers to binary.
- A. AE7.D2
  - B. 2C5.21F8
  - C. 1B6.64E

## Answers

20.  $16_{10}$

21. A.  $783 \div 16 = 48$  with remainder  $15 = F \leftarrow \text{LSD}$   
 $48 \div 16 = 3$   $0 = 0$   
 $3 \div 16 = 0$   $3 = 3 \leftarrow \text{MSD}$

$$783_{10} = 30F_{16}$$

B.  $5372_{10} = 14FC_{16}$

C.  $957_{10} = 3BD_{16}$

22. A.  $0.653 \times 16 = 10.448 = 0.448$  with overflow  $10 = A \leftarrow \text{MSD}$   
 $0.448 \times 16 = 7.168 = 0.168$   $7 = 7$   
 $0.168 \times 16 = 2.688 = 0.688$   $2 = 2$   
 $0.688 \times 16 = 11.008 = 0.008$   $11 = B \leftarrow \text{LSD}$

$$0.653_{10} = 0.A72B_{16}$$

B.  $0.109375_{10} = 0.1C_{16}$

C.  $0.4567_{10} = 0.74EA_{16}$

23. A.  $1573.125_{10} = 1573_{10} + 0.125_{10}$   
 $1573 \div 16 = 98$  with remainder  $5 = 5 \leftarrow \text{LSD}$   
 $98 \div 16 = 6$   $2 = 2$   
 $6 \div 16 = 0$   $6 = 6 \leftarrow \text{MSD}$

$$1573_{10} = 625_{16}$$

$0.125 \times 16 = 2.00 = 0$  with overflow  $2 = 2 \leftarrow \begin{matrix} \text{MSD} \\ \text{LSD} \end{matrix}$

$$0.125_{10} = 0.2_{16}$$

$$1573.125_{10} = 1573_{10} + 0.125_{10} = 625_{16} + 0.2_{16} = 625.2_{16}$$

24. A.  $100001101.01011_2 = 0001\ 0000\ 1101.0101\ 1000_2$   
 $= 10D.58_{16}$

B.  $11111011001.01_2 = 7D9.4_{16}$

C.  $110001101.00010010101_2 = 18D.12A_{16}$

25. A.  $AE7.D2_{16} = 1010\ 1110\ 0111.1101\ 0010_2$   
 $= 101011100111.1101001_2$

B.  $2C5.21F8_{16} = 1011000101.0010000111111_2$

C.  $1B6.64E_{16} = 110110110.01100100111_2$

## BINARY CODES

Converting a decimal number into its binary equivalent is called "coding." A decimal number is expressed as a binary code or binary number. The **binary number system**, as discussed, is known as the pure binary code. This name distinguishes it from other types of binary codes. This section will discuss some of the other types of binary codes used in computers.

### Binary Coded Decimal

The decimal number system is easy to use because it is so familiar. The binary number system is less convenient to use because it is less familiar. It is difficult to quickly glance at a binary number and recognize its decimal equivalent. For example, the binary number 1010011 represents the decimal number 83. It is difficult to tell immediately by looking at the number what its decimal value is. However, within a few minutes, using the procedures described earlier, you could readily calculate its decimal value. The amount of time it takes to convert or recognize a binary number quantity is a distinct disadvantage in working with this code despite the numerous hardware advantages. Engineers recognized this problem early and developed a special form of binary code that was more compatible with the decimal system. Because so many digital devices, instruments and equipment use decimal input and output, this special code has become very widely used and accepted. This special compromise code is known as binary coded decimal (BCD). The BCD code combines some of the characteristics of both the binary and decimal number systems.

**8421 BCD Code** The BCD code is a system of representing the decimal digits 0 through 9 with a four-bit binary code. This BCD code uses the standard 8421 position **weighting system** of the pure binary code. The standard 8421 BCD code and the decimal equivalents are shown in Figure 1-11, along with a special Gray code that will be described later. As with the pure binary code, you can convert the BCD numbers into their decimal equivalents by simply adding together the weights of the bit positions whereby the binary 1's occur. Note, however, that there are only ten possible valid 4-bit code arrangements. The 4-bit binary numbers representing the decimal numbers 10 through 15 are invalid in the BCD system.



| DECIMAL | 8421 BCD  | GRAY | BINARY |
|---------|-----------|------|--------|
| 0       | 0000      | 0000 | 0000   |
| 1       | 0001      | 0001 | 0001   |
| 2       | 0010      | 0011 | 0010   |
| 3       | 0011      | 0010 | 0011   |
| 4       | 0100      | 0110 | 0100   |
| 5       | 0101      | 0111 | 0101   |
| 6       | 0110      | 0101 | 0110   |
| 7       | 0111      | 0100 | 0111   |
| 8       | 1000      | 1100 | 1000   |
| 9       | 1001      | 1101 | 1001   |
| 10      | 0001 0000 | 1111 | 1010   |
| 11      | 0001 0001 | 1110 | 1011   |
| 12      | 0001 0010 | 1010 | 1100   |
| 13      | 0001 0011 | 1011 | 1101   |
| 14      | 0001 0100 | 1001 | 1110   |
| 15      | 0001 0101 | 1000 | 1111   |

Figure 1-11

Codes.

To represent a decimal number in BCD notation, substitute the appropriate 4-bit code for each decimal digit. For example, the decimal integer 834 in BCD would be 1000 0011 0100. Each decimal digit is represented by its equivalent 8421 4-bit code. A space is left between each 4-bit group to avoid confusing the BCD format with the pure binary code. This method of representation also applies to decimal fractions. For example, the decimal fraction 0.764 would be 0.0111 0110 0100 in BCD. Again, each decimal digit is represented by its equivalent 8421 4-bit code, with a space between each group.

An advantage of the BCD code is that the ten BCD code combinations are easy to remember. Once you begin to work with binary numbers regularly, the BCD numbers may come to you as quickly and automatically as decimal numbers. For that reason, by simply glancing at the BCD representation of a decimal number you can make the conversion almost as quickly as if it were already in decimal form. As an example, convert a BCD number into its decimal equivalent.

$$0110\ 0010\ 1000.1001\ 0101\ 0100 = 628.954_{10}$$

The BCD code simplifies the man-machine interface but it is less efficient than the pure binary code. It takes more bits to represent a given decimal number in BCD than it does with pure binary notation. For example, the decimal number 83 in pure binary form is 1010011. In BCD code the decimal number 83 is written as 1000 0011. In the pure binary code, it takes only seven bits to represent the number 83. In BCD form, it takes eight bits. It is inefficient because, for each bit in a data word, there is usually some digital circuitry associated with it. The extra circuitry associated with the BCD code costs more, increases equipment complexity, and consumes more power. Arithmetic operations with BCD numbers are also more time consuming and complex than those with pure binary numbers. With four bits of binary information, you can represent a total of  $2^4 = 16$  different states or the decimal number equivalents 0 through 15. In the BCD system, six of these states (10-15), are wasted. When the BCD number system is used, some efficiency is traded for the improved communications between the digital equipment and the human operator.

Decimal-to-BCD conversion is simple and straightforward. However, binary-to-BCD conversion is not direct. An intermediate conversion to decimal must be performed first. For example, the binary number 1011.01 is converted into its BCD equivalent.

First the binary number is converted to decimal.

$$\begin{aligned} 1011.01_2 &= (1 \times 2^3) + (0 \times 2^2) + (1 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) \\ &= 8 + 0 + 2 + 1 + 0 + 0.25 \\ &= 11.25_{10} \end{aligned}$$

Then the decimal result is converted to BCD.

$$11.25_{10} = 0001\ 0001.0010\ 0101$$

To convert from BCD to binary, the previous operation is reversed. For example, the BCD number 1001 0110.0110 0010 0101 is converted into its binary equivalent.

First, the BCD number is converted to decimal.

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10}$$

Then the decimal result is converted to binary.

$$96.625_{10} = 96_{10} + 0.625_{10}$$

|                  |                |   |       |
|------------------|----------------|---|-------|
| $96 \div 2 = 48$ | with remainder | 0 | ← LSB |
| $48 \div 2 = 24$ |                | 0 |       |
| $24 \div 2 = 12$ |                | 0 |       |
| $12 \div 2 = 6$  |                | 0 |       |
| $6 \div 2 = 3$   |                | 0 |       |
| $3 \div 2 = 1$   |                | 1 |       |
| $1 \div 2 = 0$   |                | 1 | ← MSB |

$96_{10} = 1100000_2$

|                                |               |   |       |
|--------------------------------|---------------|---|-------|
| $0.625 \times 2 = 1.25 = 0.25$ | with overflow | 1 | ← MSB |
| $0.250 \times 2 = 0.50 = 0.50$ |               | 0 |       |
| $0.500 \times 2 = 1.00 = 0$    |               | 1 | ← LSB |

$0.625_{10} = 0.101_2$

$$96.625_{10} = 96_{10} + 0.625_{10} = 1100000_2 + 0.101_2 = 1100000.101_2$$

Therefore:

$$1001\ 0110.0110\ 0010\ 0101 = 96.625_{10} = 1100000.101_2$$

Because the intermediate decimal number contains both an integer and fraction, each number portion is converted as described under "Binary Number System." The binary sum (integer plus fraction) 1100000.101 is equivalent to the BCD number 1001 0110.0110 0010 0101.



| DECIMAL | 8421 BCD  | GRAY | BINARY |
|---------|-----------|------|--------|
| 0       | 0000      | 0000 | 0000   |
| 1       | 0001      | 0001 | 0001   |
| 2       | 0010      | 0011 | 0010   |
| 3       | 0011      | 0010 | 0011   |
| 4       | 0100      | 0110 | 0100   |
| 5       | 0101      | 0111 | 0101   |
| 6       | 0110      | 0101 | 0110   |
| 7       | 0111      | 0100 | 0111   |
| 8       | 1000      | 1100 | 1000   |
| 9       | 1001      | 1101 | 1001   |
| 10      | 0001 0000 | 1111 | 1010   |
| 11      | 01 0001   | 1110 | 1011   |
| 12      | 01 0010   | 1010 | 1100   |
| 13      | 01 0011   | 1011 | 1101   |
| 14      | 0001 0100 | 1001 | 1110   |
| 15      | 0001 0101 | 1000 | 1111   |

Figure 1-11  
Codes.

## Special Binary Codes

Besides the standard pure binary coded form, the BCD numbering system is by far the most widely-used digital code. You will find one or the other in most of the applications that you encounter. However, there are several other codes that are used for special applications, such as the "Gray Code."

The Gray Code is a widely-used, non-weighted code system. Also known as the cyclic, unit distance or reflective code, the Gray code can exist in either the pure binary or BCD formats. The Gray code is shown in Figure 1-11. As with the pure binary code, the first ten codes are used in BCD operations. Notice that there is a change in only one bit from one code number to the next in sequence. You can get a better idea about the Gray code sequence by comparing it to the standard 4-bit 8421 BCD code and the pure binary code also shown in Figure 1-11. For example, consider the change from 7 (0111) to 8 (1000) in the pure binary code. When this change takes place, all bits change. Bits that were 1's are changed to 0's and 0's are changed to 1's. Now notice the code change from 7 to 8 in the Gray code. Here 7 (0100) changes to 8 (1100). Only the first bit changes.

The Gray code is generally known as an error minimizing code because it greatly reduces confusion in the electronic circuitry when changing from one state to the next. When binary codes are implemented with electronic circuitry, it takes a finite period of time for bits to change from 0 to 1 or 1 to 0. These state changes can create timing and speed problems. This is particularly true in the standard 8421 codes where many bits change from one combination to the next. When the Gray code is used, however, the timing and speed errors are greatly minimized because only one bit changes at a time. This permits code circuitry to operate at higher speeds with fewer errors.

The biggest disadvantage of the Gray code is that it is difficult to use in arithmetic computations. Where numbers must be added, subtracted or used in other computations, the Gray code is not applicable. In order to perform arithmetic operations, the Gray code number must generally be converted into pure binary form.



## Alphanumeric Codes

Several binary codes are called alphanumeric codes because they are used to represent characters as well as numbers. The two most common codes that will be discussed are ASCII and BAUDOT.

**ASCII Code** The American Standard Code for Information Interchange commonly referred to as ASCII, is a special form of binary code that is widely used in microprocessors and data communications equipment. A new name for this code that is becoming more popular is the American National Standard Code for Information Interchange (ANSI). However, this course will use the most recognized term, ASCII. ASCII is a 6-bit binary code that is used in transferring data between microprocessors and their peripheral devices, and in communicating data by radio and telephone. With six bits, a total of  $2^6 = 64$  different characters can be represented. These characters comprise decimal numbers 0 through 9, upper-case letters of the alphabet, plus other special characters used for punctuation and data control. A 7-bit code called full ASCII, extended ASCII, or USASCII can be represented by  $2^7 = 128$  different characters. In addition to the characters and numbers generated by 6-bit ASCII, 7-bit ASCII contains lower-case letters of the alphabet, and additional characters for punctuation and control. The 7-bit ASCII code is shown in Figure 1-12.

| COLUMN |                     | 0 <sup>(3)</sup> | 1 <sup>(3)</sup> | 2 <sup>(3)</sup> | 3   | 4   | 5                | 6   | 7 <sup>(3)</sup> |
|--------|---------------------|------------------|------------------|------------------|-----|-----|------------------|-----|------------------|
| ROW    | BITS<br>4321<br>765 | 000              | 001              | 010              | 011 | 100 | 101              | 110 | 111              |
| 0      | 0000                | NUL              | DLE              | SP               | 0   | @   | P                | \   | p                |
| 1      | 0001                | SOH              | DC1              | !                | 1   | A   | Q                | a   | q                |
| 2      | 0010                | STX              | DC2              | "                | 2   | B   | R                | b   | r                |
| 3      | 0011                | ETX              | DC3              | #                | 3   | C   | S                | c   | s                |
| 4      | 0100                | EOT              | DC4              | \$               | 4   | D   | T                | d   | t                |
| 5      | 0101                | ENQ              | NAK              | %                | 5   | E   | U                | e   | u                |
| 6      | 0110                | ACK              | SYN              | &                | 6   | F   | V                | f   | v                |
| 7      | 0111                | BEL              | ETB              | '                | 7   | G   | W                | g   | w                |
| 8      | 1000                | BS               | CAN              | (                | 8   | H   | X                | h   | x                |
| 9      | 1001                | HT               | EM               | )                | 9   | I   | Y                | i   | y                |
| 10     | 1010                | LF               | SUB              | *                | :   | J   | Z                | j   | z                |
| 11     | 1011                | VT               | ESC              | +                | ;   | K   | [                | k   | {                |
| 12     | 1100                | FF               | FS               | ,                | <   | L   | \                | l   | !                |
| 13     | 1101                | CR               | GS               | -                | =   | M   | ]                | m   | }                |
| 14     | 1110                | SO               | RS               | .                | >   | N   | ~ <sup>(1)</sup> | n   | ~                |
| 15     | 1111                | SI               | US               | /                | ?   | O   | — <sup>(2)</sup> | o   | DEL              |

Figure 1-12

Table of 7-bit American Standard Code  
for Information Interchange.



## NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) Explanation of special control functions in columns 0, 1, 2, and 7.

|     |                                              |     |                           |
|-----|----------------------------------------------|-----|---------------------------|
| NUL | Null                                         | DLE | Data Link Escape          |
| SOH | Start of Heading                             | DC1 | Device Control 1          |
| STX | Start of Text                                | DC2 | Device Control 2          |
| ETX | End of Text                                  | DC3 | Device Control 3          |
| EOT | End of Transmission                          | DC4 | Device Control 4          |
| ENQ | Enquiry                                      | NAK | Negative Acknowledge      |
| ACK | Acknowledge                                  | SYN | Synchronous Idle          |
| BEL | Bell (audible signal)                        | ETB | End of Transmission Block |
| BS  | Backspace                                    | CAN | Cancel                    |
| HT  | Horizontal Tabulation<br>(punched card skip) | EM  | End of Medium             |
| LF  | Line Feed                                    | SUB | Substitute                |
| VT  | Vertical Tabulation                          | ESC | Escape                    |
| FF  | Form Feed                                    | FS  | File Separator            |
| CR  | Carriage Return                              | GS  | Group Separator           |
| SO  | Shift Out                                    | RS  | Record Separator          |
| SI  | Shift In                                     | US  | Unit Separator            |
| SP  | Space (blank)                                | DEL | Delete                    |

Figure 1-12

(Continued.)

The 7-bit ASCII code for each number, letter or control function is made up of a 4-bit group and a 3-bit group. Figure 1-13 shows the arrangement of these two groups and the numbering sequence. The 4-bit group is on the right and bit 1 is the LSB. Note how these groups are arranged in rows and columns in Figure 1-12.

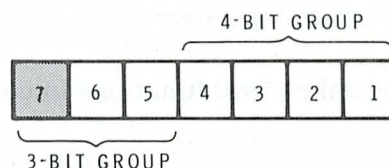


Figure 1-13  
ASCII code word format.

To determine the ASCII code for a given number letter or control operation, locate that item in the table. Then use the 3- and 4-bit codes associated with the row and column in which the item is located. For example, the ASCII code for the letter L is 1001100. It is located in column 4, row 12. The most significant 3-bit group is 100, while the least significant 4-bit group is 1100. When 6-bit ASCII is used, the 3-bit group is reduced to a 2-bit group as shown in Figure 1-14.

In 7-bit ASCII code, an eighth bit is often used as a **parity** or check bit to determine if the data (character) has been transmitted correctly. The value of this bit is determined by the type of parity desired. **Even parity** means the sum of all the 1 bits, including the parity bit, is an even number. For example, if G is the character transmitted, the ASCII code is 1000111. Since four 1's are in the code, the parity bit is 0. The 8-bit code would be written 01000111.

**OddParity** means the sum of all the 1 bits, including the parity bit, is an odd number. If the ASCII code for G was transmitted with odd parity, the binary representation would be 11000111.

| COLUMN |                    | 0                 | 1  | 2  | 3                |
|--------|--------------------|-------------------|----|----|------------------|
| ROW    | BITS<br>4321<br>65 | 10                | 11 | 00 | 01               |
| 0      | 0000               | SP <sup>(3)</sup> | 0  | @  | P                |
| 1      | 0001               | !                 | 1  | A  | Q                |
| 2      | 0010               | "                 | 2  | B  | R                |
| 3      | 0011               | #                 | 3  | C  | S                |
| 4      | 0100               | \$                | 4  | D  | T                |
| 5      | 0101               | %                 | 5  | E  | U                |
| 6      | 0110               | &                 | 6  | F  | V                |
| 7      | 0111               | '                 | 7  | G  | W                |
| 8      | 1000               | (                 | 8  | H  | X                |
| 9      | 1001               | )                 | 9  | I  | Y                |
| 10     | 1010               | *                 | :  | J  | Z                |
| 11     | 1011               | +                 | ;  | K  |                  |
| 12     | 1100               | ,                 | <  | L  | \                |
| 13     | 1101               | -                 | =  | M  |                  |
| 14     | 1110               | .                 | >  | N  | ⌒ <sup>(1)</sup> |
| 15     | 1111               | /                 | ?  | O  | — <sup>(2)</sup> |

Figure 1-14

Table of 6-bit American Standard Code  
for Information Interchange.

Appendix B

# NOTES:

- (1) Depending on the machine using this code, the symbol may be a circumflex, an up-arrow, or a horizontal parenthetical mark.
- (2) Depending on the machine using this code, the symbol may be an underline, a back-arrow, or a heart.
- (3) SP — Space (blank) for machine control.



**BAUDOT Code** While the ASCII code is used almost exclusively with microprocessor peripheral devices (CRT display, keyboard terminal, paper punch/reader, etc.), there are many older printer peripherals that use the 5-bit BAUDOT code. With five data bits, this code can represent only  $2^5 = 32$  different characters. To obtain a greater character capability, 26 of the 5-bit codes are used to represent two separate characters. As shown in Figure 1-15, one set of 5-bit codes represents the 26 upper-case alphabet letters. The same 5-bit codes also represent various figures and the decimal number series 0 through 9.

The remaining six 5-bit codes are used for machine control and do not have a secondary function. Two of these 5-bit codes determine which of the 26 double (letter/figure) characters can be transmitted/received. Bit number 11111 forces the printer to recognize all following 5-bit codes as **letters**. Bit number 11011 forces **figure** recognition of all the following 5-bit codes. For example, to type 56 NORTH 10 STREET, the following method is used.

Type — Figures 5 6 Space

Then — Letters N O R T H Space

Then — Figures 1 0 Space

Finally — Letters S T R E E T

| Bit Numbers<br>5 4 3 2 1 | Letters Case | Figures Case |
|--------------------------|--------------|--------------|
| 0 0 0 0 0                | Blank        | Blank        |
| 0 0 0 0 1                | E            | 3            |
| 0 0 0 1 0                | Line Feed    | Line Feed    |
| 0 0 0 1 1                | A            | —            |
| 0 0 1 0 0                | Space        | Space        |
| 0 0 1 0 1                | S            | Bell         |
| 0 0 1 1 0                | I            | 8            |
| 0 0 1 1 1                | U            | 7            |
| 0 1 0 0 0                | Car. Ret.    | Car. Ret.    |
| 0 1 0 0 1                | D            | \$           |
| 0 1 0 1 0                | R            | 4            |
| 0 1 0 1 1                | J            | (Apos)'      |
| 0 1 1 0 0                | N            | (Comma),     |
| 0 1 1 0 1                | F            | !            |
| 0 1 1 1 0                | C            | :            |
| 0 1 1 1 1                | K            | (            |
| 1 0 0 0 0                | T            | 5            |
| 1 0 0 0 1                | Z            | "            |
| 1 0 0 1 0                | L            | )            |
| 1 0 0 1 1                | W            | 2            |
| 1 0 1 0 0                | H            | Stop         |
| 1 0 1 0 1                | Y            | 6            |
| 1 0 1 1 0                | P            | 0            |
| 1 0 1 1 1                | Q            | 1            |
| 1 1 0 0 0                | O            | 9            |
| 1 1 0 0 1                | B            | ?            |
| 1 1 0 1 0                | G            | &            |
| 1 1 0 1 1                | Figures      | Figures      |
| 1 1 1 0 0                | M            | .            |
| 1 1 1 0 1                | X            | /            |
| 1 1 1 1 0                | V            | ;            |
| 1 1 1 1 1                | Letters      | Letters      |

Figure 1-15  
5-bit BAUDOT code table.

## Self-Test Review

26. The BCD code is more convenient to use than the binary code because:
- A. it uses less bits.
  - B. it is more compatible with the decimal number system.
  - C. it is more adaptable to arithmetic computations.
  - D. there are more different coding schemes available.
27. Convert the following decimal numbers to 8421 BCD code.
- A. 1049
  - B. 267
  - C. 835
28. Convert the following 8421 BCD code numbers to decimal.
- A. 1001 0110 0010
  - B. 0111 0001 0100 0011
  - C. 1010 1001 1000
  - D. 1000 0000 0101
29. Convert the following binary numbers to 8421 BCD code.
- A. 101110.01
  - B. 1001.0101
  - C. 11011011.0001



30. Convert the following 8421 BCD codes to binary.
- A. 0001 1000 0010.0101
  - B. 0010 1001 0000.0010 0101
  - C. 1101 0110 0011.0101
  - D. 0110 1000.0001 0010 0101
31. Which code is best for error minimizing?
- A. 8421 BCD
  - B. pure binary
  - C. Gray
32. The ASCII and BAUDOT codes are a form of \_\_\_\_\_ codes.
33. To determine if the correct ASCII character has been transmitted, a \_\_\_\_\_ bit is often added to the code.
34. Which type of parity is used when the 8-bit ASCII character 01000111 is transmitted?
- A. odd
  - B. even
35. Refer to Figure 1-12 and convert the following characters into their ASCII 7-bit binary code.
- A. B
  - B. X
  - C. 3
  - D. S
36. Refer to Figure 1-12 and convert the following ASCII 7-bit binary codes to their character equivalents.
- A. 0110010
  - B. 1010110
  - C. 1011010
  - D. 1001110

## Answers

26. B. More compatible with the decimal system.
27. A. 0001 0000 0100 1001  
B. 0010 0110 0111  
C. 1000 0011 0101
28. A. 962  
B. 7143  
C. Invalid (1010 represents a decimal digit greater than 9)  
D. 805
29. A.  $101110.01_2 = (1 \times 2^5) + (0 \times 2^4) + (1 \times 2^3) + (1 \times 2^2) + (1 \times 2^1)$   
 $+ (0 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2})$   
 $= 32 + 0 + 8 + 4 + 2 + 0 + 0 + 0.25$   
 $= 46.25_{10}$   
 $46.25_{10} = 0100\ 0110.0010\ 0101$   
 $101110.01_2 = 0100\ 0110.0010\ 0101$   
B.  $1001.0101_2 = 1001.0011\ 0001\ 0010\ 0101$   
C.  $11011011.0001_2 = 0010\ 0001\ 1001.0000\ 0110\ 0010\ 0101$





## EXPERIMENTS 1 AND 2

Perform Experiments 1 and 2 in the Experiment Section, Unit 9 of the course. After you finish the experiment, return to this unit and complete the "Unit Examination."

## UNIT EXAMINATION

This examination will test your knowledge of the important facts in this unit. It will tell you what you have learned and what you need to review. Answer all questions first; then check your work against the correct answers given later.

1. Indicate the base or radix of the following number systems.

- A. Octal \_\_\_\_\_.
- B. Decimal \_\_\_\_\_.
- C. Hexadecimal \_\_\_\_\_.
- D. Binary \_\_\_\_\_.

2. Write the following numbers using positional notation.

- A.  $1101.011_2$
- B.  $1010.01_{10}$
- C.  $1001.101_8$ .
- D.  $1110.11_{16}$

3. Convert the following numbers to decimal.

- A.  $10011.011_2$
- B.  $752.31_8$
- C.  $A8C.5F_{16}$

4. Convert the following numbers to binary.

- A.  $105.0625_{10}$
- B.  $374.24_8$
- C.  $F19.6C_{16}$

5. Convert the following numbers to octal.

- A.  $638.3125_{10}$
- B.  $10010101.0110101_2$

6. Convert the following numbers to hexadecimal.

- A.  $9587.03125_{10}$
- B.  $1101101101010.101110101_2$

7. The ASCII and BAUDOT codes are a form of \_\_\_\_\_ codes.
8. Convert the following numbers to 8421 BCD code.
  - A.  $521.372_{10}$
  - B.  $1010.011_2$
9. Convert the 8421 BCD code 1001 0101.0111 0011 to decimal.
10. Convert the 8421 BCD code 0101 0011.0111 0101 to binary.
11. Which type of parity is used when the 8-bit ASCII character 01110111 is transmitted?
  - A. Odd
  - B. Even
12. The BAUDOT code uses \_\_\_\_\_ bit numbers to generate a character.
13. Using only your knowledge of binary codes, identify the Gray code.

| Decimal | a    | b    | c    | d    |
|---------|------|------|------|------|
| 0       | 0000 | 0000 | 0000 | 0011 |
| 1       | 0001 | 0001 | 0001 | 0100 |
| 2       | 0011 | 0010 | 0010 | 0101 |
| 3       | 0010 | 0011 | 0011 | 0110 |
| 4       | 0110 | 0100 | 0100 | 0111 |
| 5       | 0111 | 0101 | 1011 | 1000 |
| 6       | 0101 | 0110 | 1100 | 1001 |
| 7       | 0100 | 0111 | 1101 | 1010 |
| 8       | 1100 | 1000 | 1110 | 1011 |
| 9       | 1101 | 1001 | 1111 | 1100 |



## EXAMINATION ANSWERS

1. A. 8  
B. 10  
C. 16  
D. 2
2. A.  $(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) + (0 \times 2^{-1}) + (1 \times 2^{-2}) + (1 \times 2^{-3})$   
B.  $(1 \times 10^3) + (0 \times 10^2) + (1 \times 10^1) + (0 \times 10^0) + (0 \times 10^{-1}) + (1 \times 10^{-2})$   
C.  $(1 \times 8^3) + (0 \times 8^2) + (0 \times 8^1) + (1 \times 8^0) + (1 \times 8^{-1}) + (0 \times 8^{-2}) + (1 \times 8^{-3})$   
D.  $(1 \times 16^3) + (1 \times 16^2) + (1 \times 16^1) + (0 \times 16^0) + (1 \times 16^{-1}) + (1 \times 16^{-2})$
3. A.  $19.375_{10}$   
B.  $490.390625_{10}$   
C.  $2700.37109375_{10}$
4. A.  $1101001.0001_2$   
B.  $11111100.0101_2$   
C.  $111100011001.011011_2$
5. A.  $1176.24_8$   
B.  $225.324_8$
6. A.  $2573.08_{16}$   
B.  $1B6A.BA8_{16}$
7. Alpha numeric
8. A. 0101 0010 0001.0011 0111 0010  
B. 0001 0000.0011 0111 0101
9.  $95.73_{10}$
10.  $110101.11_2$
11. B. Even
12. 5
13. A. Gray code



**APPENDIX C**

**BENTON HARBOR BASIC**

**and**

**EXTENDED BENTON HARBOR  
BASIC**

This Appendix is a typical "User's Manual", as supplied to describe the features of a particular version of BASIC.

This BENTON HARBOR BASIC User's Manual illustrates the scope of the special instructions you can expect from such a user's Manual, and will further illustrate the features of BENTON HARBOR BASIC, which we have used for our course material.



## TABLE OF CONTENTS

### INTRODUCTION

|                                                            |      |
|------------------------------------------------------------|------|
| Manual Scope .....                                         | 17-6 |
| Hardware Requirements .....                                | 17-6 |
| Loading and Running BASIC .....                            | 17-7 |
| Benton Harbor BASIC and Extended Benton Harbor BASIC ..... | 17-7 |
| Command Completion .....                                   | 17-7 |

### BASIC ARITHMETIC

|                             |       |
|-----------------------------|-------|
| Data Types .....            | 17-9  |
| Variables .....             | 17-11 |
| Subscripted Variables ..... | 17-12 |
| Expressions .....           | 17-14 |
| Arithmetic Operators .....  | 17-14 |
| Relational Operators .....  | 17-18 |
| Boolean Operators .....     | 17-19 |

### STRING MANIPULATION

|                        |       |
|------------------------|-------|
| String Variables ..... | 17-21 |
| String Operators ..... | 17-22 |

### THE COMMAND MODE

|                                                      |       |
|------------------------------------------------------|-------|
| Using The Command Mode For Statement Execution ..... | 17-23 |
|------------------------------------------------------|-------|

### BASIC STATEMENTS

|                                                       |       |
|-------------------------------------------------------|-------|
| Line Numbers .....                                    | 17-25 |
| Statement Types .....                                 | 17-25 |
| Command Mode Statements .....                         | 17-27 |
| Statements Valid In the Command or Program Mode ..... | 17-33 |
| Program Mode Statements .....                         | 17-58 |

### PREDEFINED FUNCTIONS

|                                                |       |
|------------------------------------------------|-------|
| Introduction .....                             | 17-61 |
| Arithmetic and Special Feature Functions ..... | 17-61 |
| STRING Functions (Extended BASIC only) .....   | 17-68 |

### EDITING COMMANDS

|                              |       |
|------------------------------|-------|
| Control-C, CNTRL-C .....     | 17-71 |
| Inputting Control .....      | 17-71 |
| Outputting Control .....     | 17-72 |
| Command Completion .....     | 17-72 |
| Enforced Lexical Rules ..... | 17-73 |
| General Text Rules .....     | 17-73 |

## ERRORS

|                              |       |
|------------------------------|-------|
| Error Messages .....         | 17-75 |
| Recovering from Errors ..... | 17-75 |

|                         |       |
|-------------------------|-------|
| BASIC ERROR TABLE ..... | 17-77 |
|-------------------------|-------|

## APPENDIX A

|                                                   |       |
|---------------------------------------------------|-------|
| Loading from the Software Distribution Tape ..... | 17-81 |
| Loading from a Configured Tape .....              | 17-82 |

## APPENDIX B

|                                           |       |
|-------------------------------------------|-------|
| Numeric Data .....                        | 17-83 |
| Boolean Data .....                        | 17-83 |
| String Data (Extended Basic Only) .....   | 17-83 |
| Variables .....                           | 17-83 |
| Sunscripted Variables .....               | 17-84 |
| Arithmetic Operators .....                | 17-84 |
| Relational Operators .....                | 17-84 |
| Boolean Operators .....                   | 17-85 |
| String Variables .....                    | 17-85 |
| String Operators .....                    | 17-85 |
| Line Numbers .....                        | 17-85 |
| The Command Mode .....                    | 17-85 |
| Multiple Statements on One Line .....     | 17-85 |
| Command Mode Statements .....             | 17-86 |
| Command and Program Mode Statements ..... | 17-87 |
| Program Mode Statements .....             | 17-90 |
| Predefined Functions .....                | 17-91 |
| Editing Commands .....                    | 17-93 |

## APPENDIX C

|                              |       |
|------------------------------|-------|
| BASIC Utility Routines ..... | 17-95 |
|------------------------------|-------|

## APPENDIX D

|                                        |        |
|----------------------------------------|--------|
| Entry Points to Utility Routines ..... | 17-107 |
|----------------------------------------|--------|

## APPENDIX E

|                         |       |
|-------------------------|-------|
| An Example of USR ..... | 7-111 |
|-------------------------|-------|

|             |        |
|-------------|--------|
| INDEX ..... | 17-113 |
|-------------|--------|



## TAB GUIDE

|                            |  |
|----------------------------|--|
| BASIC ARITHMETIC .....     |  |
| STRING MANIPULATION .....  |  |
| THE COMMAND MODE .....     |  |
| BASIC STATEMENTS .....     |  |
| PREDEFINED FUNCTIONS ..... |  |
| EDITING COMMANDS .....     |  |
| ERRORS .....               |  |
| BASIC ERROR TABLE .....    |  |
| APPENDIX A .....           |  |
| APPENDIX B .....           |  |
| APPENDIX C .....           |  |
| APPENDIX D .....           |  |
| APPENDIX E .....           |  |



## INTRODUCTION

BENTON HARBOR BASIC is a conversational programming language which is an adaptation of Dartmouth BASIC\*. (BASIC is an acronym for Beginners' All Purpose Symbolic Instruction Code.) It uses simple English statements and familiar algebraic equations to perform an operation or a series of operations to solve a problem. BENTON HARBOR BASIC is an interpretive language, compact enough to run in a Heath H8 computer with minimal memory, yet powerful enough to satisfy most problem-solving requirements. The interpretive structure of BASIC affords excellent facilities for the detection and correction of programming errors. It uses advanced techniques to perform intricate manipulations and to express problems more efficiently.

Two versions of BENTON HARBOR BASIC are available. Extended BENTON HARBOR BASIC (EX. B.H. BASIC) with strings provides character string manipulation and advanced functions. BENTON HARBOR BASIC (B.H. BASIC) does not have strings and some advanced functions, and so uses less memory. The user may operate B.H. BASIC in an H8 computer with 8K of memory.

### Manual Scope

This Manual is written for the user who is already familiar with the language BASIC. It also describes the extended implementation of Dartmouth BASIC and, in so doing, provides a brief summary of the language. However, this manual is not intended as an instruction Manual for the language BASIC. If you are not familiar with BASIC, we suggest that you obtain the Heathkit Continuing Education course entitled "Basic Programming," Model EC-1100, before attempting to use this Manual.

### Hardware Requirements

Extended BENTON HARBOR BASIC (with strings) runs on an H8 computer with a minimum of 12K bytes of random access memory. BENTON HARBOR BASIC (without strings) runs on an H8 with a minimum of 8K memory. Both versions require a console terminal, its appropriate interface card, and a mass storage device such as a cassette or paper tape reader/punch.

Both BASICs automatically measure the maximum amount of unbroken memory above the starting point at 8K (40,100 offset octal). They use all available memory unless the high memory limit is configured otherwise during the system configuration procedure (see "Product Installation" on Page 0-19 in the "Introduction" to this Software Reference Manual).

\*BASIC is a registered trademark of the Trustees of Dartmouth College.

## Benton Harbor BASIC and Extended Benton Harbor BASIC

This Manual covers both BENTON HARBOR BASIC (B.H. BASIC) and Extended BENTON HARBOR BASIC (EX. B.H. BASIC). Information that applies only to Extended BASIC is printed in a different type face, as shown below. For example:

*Strings are only used in EX. B.H. BASIC, except in PRINT statements.*

Anything not marked in this different type face applies to BASIC. References to "BASIC" apply to both BENTON HARBOR BASIC and to Extended BENTON HARBOR BASIC. BASIC is summarized in "Appendix B."

## Loading and Running BASIC

BASIC is distributed in binary load format on cassette tapes or paper tape. It is loaded in accordance with the software configuration guide, outlined in "Product Installation" on Page 0-19. A condensed version of this loading procedure is given in "Appendix A" of this Manual. Once a system BASIC tape is configured, you can load the configured tape, using the internal PAM-8 loader, and start it by pressing the GO key. EX. B.H. BASIC and B.H. BASIC use the asterisk (\*) as a prompt character.

## Command Completion

Both the B.H. BASIC and EX. B.H. BASIC employ command completion. BASIC examines each character as you type it on the console keyboard, and when sufficient information is received to uniquely identify one particular command, BASIC finishes typing the command for you. For example, once the letters PR have been typed, the command PRINT is uniquely defined. Therefore, BASIC supplies letters INT and the required blank following the T.

BASIC also watches your spelling. As commands are being typed, letter combinations leading to non-existent commands are not accepted and the console terminal bell is rung.

## Bentley Harbor BASIC and Extended Bentley Harbor BASIC

The Bentley Harbor BASIC and Extended Bentley Harbor BASIC (B.H. BASIC and E.B. BASIC) are the only BASICs that apply only to the Bentley Harbor BASIC and Extended Bentley Harbor BASIC. Information that applies only to the Bentley Harbor BASIC and Extended Bentley Harbor BASIC is shown below. For example, the following information is not shown below.

The only cases in EX B.H. BASIC except in PRINT statements.

The only cases in EX B.H. BASIC except in PRINT statements. The only cases in EX B.H. BASIC except in PRINT statements. The only cases in EX B.H. BASIC except in PRINT statements. The only cases in EX B.H. BASIC except in PRINT statements.

## Loading and Running BASIC

The BASIC is distributed in three hard format on cassette tapes or paper tape. It is distributed with the software configuration. It is distributed in three hard format on cassette tapes or paper tape. It is distributed with the software configuration. It is distributed in three hard format on cassette tapes or paper tape. It is distributed with the software configuration.

## Command Completion

The BASIC and EX B.H. BASIC empty command completion. BASIC and EX B.H. BASIC empty command completion. BASIC and EX B.H. BASIC empty command completion. BASIC and EX B.H. BASIC empty command completion.

The BASIC and EX B.H. BASIC empty command completion. BASIC and EX B.H. BASIC empty command completion. BASIC and EX B.H. BASIC empty command completion. BASIC and EX B.H. BASIC empty command completion.



## BASIC ARITHMETIC

### Data Types

BASIC supports three different data types:

1. Numeric data.
2. Boolean data.
3. String data.

#### NUMERIC DATA

BASIC accepts real and integer numbers. A real number contains a decimal point. BASIC assumes a decimal point **after** integer data. Any number can be used in mathematical expression without regard to its type. Real numbers must be in the approximate range of  $10^{-38}$  to  $10^{+37}$ . Integer numbers must lie in the range of 0 to 65535. All numbers used in BASIC are internally represented in floating point, which allows approximately 6.9 digits of accuracy. Numbers may be either negative or positive.

In addition to integer and real numbers, BASIC recognizes a third format. This format, called exponential notation, expresses a number as a decimal number raised to a power of 10. The exponential form is

$XXE(\pm)NN$

where E represents the algebraic statement "times ten to the power of," XX represents up to a six digit integer or real number, and NN represents an integer from 0 to 38. Thus, the number is read as "XX times 10 to the  $\pm$  power of NN."

Numeric data in all three forms may be used in the immediate mode, program mode in data statements, or in response to READ and INPUT statements.

Unless otherwise specified, all the numbers including exponents are presumed to be positive.



The results of BASIC computations are printed as decimal numbers if they lie in the range of 0.1 to 999999\*. If the results do not fall in this range, the exponential format is used. BASIC automatically suppresses all leading and trailing zeros in real and integer numbers. When the output is in exponential format, it is in the form

( $\pm$ ) X.XXXXXE ( $\pm$ ) NN

The following are examples of typical inputs and the corresponding output. Note the dropping of leading and trailing zeros, truncation to six places of accuracy, conversion to exponential notation when necessary, and conversion to decimal notation where permitted.

| <u>INPUT NUMBER</u> | <u>OUTPUT NUMBER</u> | <u>COMMENTS</u>               |
|---------------------|----------------------|-------------------------------|
| 0.1                 | .1                   | (leading zero dropped)        |
| .0079               | 7.90000E-03          | (<.1 converts to exponential) |
| 0022                | 22                   | (leading zeros dropped)       |
| 22.0200             | 22.02                | (trailing zeros dropped)      |
| 999999              | 999999               | (format maintained)           |
| 1000000             | 1.00000E+06          | (converted to exponential)    |
| 100000007           | 1.00000E+08          | (truncated to 6 places)       |
| -10.1E+2            | -1010                | (converted to decimal format) |

## BOOLEAN VALUES

Boolean values are a subclass of numeric values. Values representing the positive integers from 0-65,535 ( $2^{16}-1$ ) may be used as Boolean data. When using numeric data as Boolean values, the numeric data represents the equivalent 16-bit binary numbers. Fractional parts of numeric data used with Boolean operators are discarded. If the numeric value with the fractional part does not fall into the range of 0-65,535, an illegal number error is generated.

## STRING DATA (Extended BASIC Only)

*Extended BASIC handles data in a character string format. Data elements of this type are made up of a string of ASCII characters up to 255 characters in length. Extended BASIC provides operators and functions to manipulate string data. String values in either programmed text or data must always be enclosed by quotation marks (""). Any printable ASCII character (with the exception of the quotation mark itself) may appear in an Extended BASIC string. In addition to the printable ASCII characters, the line feed and bell characters are also permitted. A string may not be typed on more than one line. A carriage return is rejected as an illegal string character.*

\*NOTE: This may be changed in EX. B.H. BASIC. See "CNTRL 1," Page 17-35.

## Variables

A BASIC variable is an algebraic symbol representing a number. Variable naming adheres to the Dartmouth specification. That is, variable names consist of one alphabetic character which may be followed by one digit (zero to nine). The following is a list of acceptable and unacceptable variables, and the reason why the variable is unacceptable.

| <u>ACCEPTABLE<br/>VARIABLES</u> | <u>UNACCEPTABLE<br/>VARIABLES</u> | <u>REASON FOR<br/>UNACCEPTABILITY</u>                     |
|---------------------------------|-----------------------------------|-----------------------------------------------------------|
| C                               | 2C                                | A digit cannot begin a variable.                          |
| A5                              | AF                                | A second character in a variable must be a number (0-9).  |
| D                               | 3                                 | A single number is not an acceptable variable.            |
| L2                              | \$2                               | The first character of a variable must be a letter (A-Z). |

*Subscripted variables, string variables, and subscripted string variables are permitted. See "Subscripted Variables," Page 17-12, and "String Manipulation" on Page 17-21.*

A value is assigned to a variable when you indicate the value in a LET, READ, or INPUT statement. These operations are discussed in "LET" (Page 17-46), "PRINT" (17-50), and "INPUT AND LINE INPUT" (Page 17-59).

The value assigned to a variable changes each time a statement equates the variable to a new value. The RUN command sets all variables to zero (0). Therefore, it is only necessary to assign an exact value to a variable when an initial value other than zero is required.

## Subscripted Variables

In addition to the variables described above, BASIC permits subscripted variables. Subscripted variables are of the form:

$$A_n (N_1, \dots, N_8),$$

where A is the variable letter, n is a number (optional) 0-9, and  $N_1$  thru  $N_8$  are the integer dimensions of the variable. Subscripted variables provide you with the ability to manipulate lists, tables, matrices, or any set of variables. Variables are allowed one to eight subscripts.

The use of subscripts permits you to create multi-dimensional arrays of numeric and string variables. It is important to note that a dimensioned variable is distinguished from a scalar value of the same name. For example, all four of the following are distinct variables:

$$A, A(N), A\$*, A\$(N)*$$

When you are referencing a subscripted variable, each element in the subscript list may consist of an arbitrarily complex expression so long as it evaluates to a numeric value within the allowable range for the indicated dimension. Thus, the subscripted variable A(5,5), would be dimensioned as:

|                                          |                                        |
|------------------------------------------|----------------------------------------|
| $X = A(2,3)$                             | is legal                               |
| $X = A(2 \uparrow 2, \text{VAL}("4.0"))$ | is legal as it is equivalent to A(4,4) |
| $X = A(2, "4.0")$                        | is not legal as ("4.0" is a string)    |

\*NOTE: The \$ indicates a string variable valid only in EX. B.H. BASIC. See Section 3.







BASIC does not presume any dimension. Therefore, the DIMension (DIM) statement must be used to define the maximum number of elements in any array. It is described in "DIM (DIMENSION)" on Page 17-36.

## Expressions

An expression is a group of symbols to be evaluated by BASIC. Expressions are composed of numeric data, Boolean data, string data, variables, or functions. In an expression, these are alone or combined by arithmetic, relational, or Boolean operators.

The following examples show some expressions BASIC recognizes.

| ARITHMETIC<br>EXPRESSIONS | BOOLEAN<br>EXPRESSIONS | STRING<br>EXPRESSIONS | DESCRIPTION |
|---------------------------|------------------------|-----------------------|-------------|
| 1.02                      | 255                    | "YES"                 | Data        |
| 1.02 + 16                 | 255 OR 003             | "YES" + "NO"          | Combined    |
| A < B                     |                        | "YES" < "NO"          | Relational  |

A major feature of BASIC is its extensive use of expressions in situations when many other BASICs only permit variables or numbers. This feature permits you to perform very sophisticated operations within a particular command or function. It is important to note that not all expressions can be used in all statements. The explanations describing the individual statements detail any limitations.

## Arithmetic Operators

BASIC performs *exponentiation* (in EX. B.H. BASIC only), multiplication, division, addition, and subtraction. BASIC also supports two unary operators (– and NOT). The asterisk (\*) is used to signify multiplication and the slash (/) is used to indicate division. *Exponentiation is indicated by the up arrow (↑).*

### THE PRIORITY OF ARITHMETIC OPERATIONS

When multiple operations are to be performed in a single expression, an order of priority is observed. The following list shows the arithmetic operators in order of descending precedence. Operators appearing on the same line are of equal precedence.

|          |                              |
|----------|------------------------------|
| –(Unary) | (negation)                   |
| ↑        | ( <i>exponentiation</i> )    |
| * /      | (multiplication    division) |
| + –      | (addition    subtraction)    |

Parentheses are used to change the precedence of any arithmetic operations, as they are in common algebra. Parentheses receive top priority. Any expression within parentheses is evaluated before an expression without parentheses. The innermost leftmost parenthetical expression has the greatest priority.

## UNARY OPERATORS

BASIC supports two unary operators:  $-$  and NOT. These operators are referred to as unary because they require only one operand. For example:

```
A = -2
C = NOT D
```

The unary operator ( $-$ ) performs arithmetic negation. The NOT operator performs Boolean negation. See Page 17-19.

## EXPONENTIATION (EXTENDED BASIC ONLY)

Exponentiation ( $\uparrow$ ) is used to raise numeric or variable data to a power. For example:

$A = B \uparrow 2$  is equivalent to  $A = B * B$ .

**NOTE:** The operand must not be negative. The exponent may be negative. A negative operand generates a syntax error. For greatest efficiency,  $B \uparrow 2$  should be written as  $B * B$  and  $B \uparrow 3$  should be written as  $B * B * B$ . All other powers should use the  $\uparrow$ .

## MULTIPLICATION AND DIVISION

BASIC uses the asterisk ( $*$ ) and the slash ( $/$ ) as symbols to perform the algebraic operations of multiplication and division respectively. Both multiplication and division require numeric data as operands.

The following examples use the multiplication and division operators:

```
*PRINT 2*6
```

```
12
```

```
*PRINT 6/3
```

```
2
```

```
*PRINT 6/3*2
```

```
4
```

```
*
```

**NOTE:** This last expression evaluates to 4, not 1; as  $*$  and  $/$  have equal precedence and therefore the leftmost operator is evaluated first.

## ADDITION AND SUBTRACTION

The plus sign (+) and the minus sign (−) perform arithmetic addition and subtraction. *In addition, the plus operator (+) performs string concatenation if both operands are string data. Concatenation is restricted to Extended BASIC.* The following examples use the plus and minus operators:

```
*PRINT 3
3
*PRINT 3+5
8

*PRINT 10-3
7

*PRINT "HEATH" + " " + "H8"
HEATH H8
*
} extended BASIC only
```

## SUMMARY

In any given expression, BASIC performs arithmetic operations in the following order:

1. Parentheses have top priority. Any expression in parentheses is evaluated prior to a nonparenthetical expression.
2. Without parentheses, the order of priority is:
  - a. Unary minus and NOT (equal priority).
  - b. *Exponentiation (proceeds from left to right).*
  - c. Multiplication and division (equal priority, proceeds from left to right).
  - d. Addition and subtraction (equal priority, proceeds from left to right).
3. If the rules in either 1 or 2 do not clearly designate the order of priority, the evaluation of expression proceeds from left to right.

The following examples illustrate these principles. *The expression  $2\uparrow 3\uparrow 2$  is evaluated from left to right:*

1.  $2\uparrow 3 = 8$  (left-most exponentiation has highest priority).
2.  $8\uparrow 2 = 64$  (answer).



The expression  $12/6*4$  is evaluated from left to right since multiplication and division are of equal priority:

1.  $12/6 = 2$  (division is the left-most operator).
2.  $2*4 = 8$  (answer).

The expression  $6+4*3\uparrow 2$  evaluates as:

1.  $3\uparrow 2 = 9$  (exponentiation has highest priority).
2.  $9*4 = 36$  (multiplication has second priority).
3.  $36+6 = 42$  (addition has lowest priority; answer).

Parentheses may be nested, (enclosed by additional sets of parentheses). The expression in the innermost set of parentheses is evaluated first. The next innermost left justified is second, and so on, until all parenthetical expressions are evaluated. For example:

$$6 * ((2\uparrow 3 + 4) / 3)$$

Evaluates as:

1.  $2\uparrow 3 = 8$  (exponentiation in parentheses has highest priority).
2.  $8+4 = 12$  (addition in parentheses has next highest priority).
3.  $12/3 = 4$  (next innermost parentheses are evaluated).
4.  $4*6 = 24$  (multiplication outside of parentheses is lowest priority).

Parentheses prevent confusion or doubt when you are evaluating the expression. For example, the two expressions

$$D * E \uparrow 2 / 4 + E / C * A \uparrow 2$$

$$((D * (E \uparrow 2)) / 4) + ((E / C) * (A \uparrow 2))$$

are executed identically. However, the second is much easier to understand.

Blanks should be used in a similar manner, as BASIC ignores blanks (except when they are part of a string enclosed in quotation marks). The two statements:

```
10 LET B = 3 * 2 + 1
10 LET B=3*2+1
```

are identical. The blanks in the first statement make it easier to read.



## Relational Operators

Relational operators compare two variables or expressions. They are generally used with an IF THEN statement. The result of a comparison by the relational operators is either a true or a false. A false is represented by zero, and true is represented by 65535 ( $2^{16}-1$ ). NOTE: These values are chosen so when they are used as Boolean values, false is all zeros and true is all ones.

The following table lists relational operators as used in BASIC.

| ALGEBRATIC<br>SYMBOL | BASIC<br>SYMBOL | EXAMPLE | MEANING                          |
|----------------------|-----------------|---------|----------------------------------|
| =                    | =               | A=B     | A is equal to B.                 |
| <                    | <               | A<B     | A is less than B.                |
| ≤                    | <=              | A<=B    | A is less than or equal to B.    |
| >                    | >               | A>B     | A is greater than B.             |
| >                    | >=              | A>=B    | A is greater than or equal to B. |
| ≠                    | <>              | A<>B    | A is not equal to B.             |

The symbols =<, =>, >< are not accepted and BASIC generates a syntax error if they are used.

The following examples show the results of using relational operators.

```
*PRINT 3<4 (true)
65535
```

```
*PRINT 4<3 (false)
0
```

EX. B.H. BASIC and B.H. BASIC differ from most other versions in the use of the relational operator. When you are using BASIC, you may use the relational operators in any expression. When the expression is evaluated, the appropriate numeric answer (0 or 65535) will be used as the answer to that expression.

## Boolean Operators

### OR

The operator OR performs a Boolean OR on the two integer operands. The integer operands (which must lie in the range of 0 to 65535) are converted to 16-bit binary numbers. The Boolean (logical) 16-bit OR is applied and the result is returned to the equivalent integer representation. NOTE: As the Boolean value chosen to represent true (65535) and false (0), the OR operator implements a standard truth table OR function. For example:

|                   |                          |     |
|-------------------|--------------------------|-----|
| *PRINT 132 OR 255 | 00000000 10000100        | 132 |
| 255               | 00000000 11111111        | 255 |
|                   | <u>00000000 11111111</u> | 255 |

and

```
*PRINT (3>2) OR (4>9)
65535
```

### AND

The AND operator performs a Boolean (logical) AND on the two integer operands. These integer operands must lie in the range of 0 to 65535. The integer operands are converted into 16-bit binary numbers and the logical AND is performed. The result is returned to the equivalent integer representation. NOTE: The AND operator implements a standard AND truth table on the values true (65535) AND false (0). For example:

|                    |                          |     |
|--------------------|--------------------------|-----|
| *PRINT 132 AND 255 | 00000000 10000100        | 132 |
| 132                | 00000000 11111111        | 255 |
| *                  | <u>00000000 10000100</u> | 132 |

and

```
*PRINT (3>2) AND (9>7)
65535
```

### NOT

The NOT operator Boolean negation. That is, the numeric value of the variable is converted into a 16-bit Boolean data value; each bit is inverted, and the 16-bit binary number is restored to numeric data. For example:

|              |                           |     |
|--------------|---------------------------|-----|
| *PRINT NOT 0 | 0 = 00000000 00000000     | and |
| 65535        | 65535 = 11111111 11111111 |     |
| *            |                           |     |



## STRING MANIPULATION

Extended BENTON HARBOR BASIC is capable of manipulating string information. A string is a sequence of characters treated as a single unit of an expression. It can be composed of alphanumeric and other printing characters. An alphanumeric string contains letters, numbers, blanks, or any combination of these characters. A character string may not exceed 255 characters. The blank, bell, and line feed are considered to be printing characters.

### String Variables

The dollar sign (\$) following a variable name indicates a string variable. For example:

B\$

and

L6\$

are string variables. A string variable (B\$) is used in the following example.

```
*B$ = "HI": PRINT B$
```

HI

NOTE: The string variable B\$ is separate and distinct from the variable B.

Any array name followed by the \$ character notes that the dimensioned variable is a string. For example:

|          |           |                                          |
|----------|-----------|------------------------------------------|
| L\$(n)   | A2\$(n)   | (single-dimensional string variables).   |
| D\$(m,n) | H1\$(m,n) | (multiple-dimensional string variables). |

The numbers in parentheses indicate the location within the array. See "Subscripted Variables," Page 17-12.

The same variable can be used as a numeric variable and as a string variable in one program. For example, each of the following is a different variable:

|     |          |
|-----|----------|
| B   | B(n)     |
| B\$ | B\$(m,n) |

The following are illegal, as they are double declarations of the same variable.

|        |          |
|--------|----------|
| A\$(n) | A\$(n,m) |
|--------|----------|

String arrays are defined with a dimension (DIM) statement in the same way numerical arrays are defined.



## String Operators

Extended BASIC provides you with the ability to manipulate strings. The string manipulation operators are: plus (+), for concatenation, and the relational operators.

### CONCATENATION

Concatenation connects one string to another without any intervening characters. This is specified by using the plus (+) symbol and only works with strings. The maximum range of a concatenated string is 255 characters. For example:

```
*PRINT "THE HEATH" + " H8 COMPUTER"
THE HEATH H8 COMPUTER
```

### RELATIONAL OPERATORS FOR STRINGS

Relational operators, when applied to strings, indicate alphabetic sequence. The relational comparison is done on the basis of the ASCII value associated with each character, on a character-by-character basis, using the ASCII collating sequence. A null character (indicating that the string is exhausted) is considered to head the collating sequence. For example:

```
*PRINT "ABC" < "DEF"
65536 (The relation shown is true)
*PRINT "ABC">"ABCD"
0 (The relation is false. "ABC" is less than "ABCD.")
```

**NOTE:** In any string comparison, trailing blanks are not ignored. For example:

```
*PRINT "CDE" = "CDE "
0 (The equality is false.)
```

The following table indicates how relational operators are used with string variables in Extended BASIC.

| OPERATOR | EXAMPLE    | MEANING                                          |
|----------|------------|--------------------------------------------------|
| =        | A\$ = B\$  | String A\$ and B\$ are alphabetically equal.     |
| <        | A\$ < B\$  | String A\$ is alphabetically less than B\$.      |
| >        | A\$ > B\$  | String A\$ is alphabetically greater than B\$.   |
| <=       | A\$ <= B\$ | String A\$ is equal to or less than B\$.         |
| >=       | A\$ >= B\$ | String A\$ is equal to or greater than B\$.      |
| <>       | A\$ <> B\$ | String A\$ and B\$ are not alphabetically equal. |

## THE COMMAND MODE

### Using The Command Mode For Statement Execution

You may solve a problem in BASIC by using a complete program or by use of the **command** mode. **Command** mode makes BASIC an extremely powerful calculator.

Lines of program material entered for later execution are identified by line numbers. BASIC identifies those lines entered for immediate execution by the absence of the line number. That is to say, statements that begin with line numbers are stored, and statements without line numbers are executed immediately when a carriage return is received. For example\*:

```
10 PRINT "THIS IS AN H8 COMPUTER"
```

is not executed when it is entered at the console terminal. However, the statement:

```
*PRINT "THIS IS THE HEATH H8 COMPUTER"
```

when the RETURN key is typed, is immediately executed as:

```
THIS IS THE HEATH H8 COMPUTER
```

The **command** mode of operation is useful in program de-bugging and performing simple calculations which do not justify the writing of a complete program.

For example, in order to facilitate program de-bugging, you may place **STOP** statements liberally throughout a program. Once BASIC encounters a **STOP** statement, the program halts. You can examine and change data values using the **command** mode. The statement

```
CONTINUE
```

is used to continue execution of the program. You can also use the **GOSUB** and **IF** commands. Values assigned to variables remain intact using this technique. A **SCRATCH**, **CLEAR**, or another **RUN** command resets these values.

\*NOTE: Strings may be used in B.H. BASIC PRINT statement. However, these strings cannot be manipulated in B.H. BASIC.

The ability to place multiple statements on a single line is an advantage in the **command** mode. For example:

```
*B = 2:PRINT B:PRINT B + 1
2
3
*
```

Program loops are allowed in the **command** mode. For example, a table of squares can be produced as follows:

```
*FOR A = 1 TO 10:PRINT A,A * A:NEXT A
1 1
2 4
3 9
4 16
5 25
6 36
7 49
8 64
9 81
10 100
*
```

Some statements cannot be used in the **command** mode. The INPUT statement, for example, is not available in the **command** mode, and its use results in the USE error message. There are certain command functions in the **command** mode which make no sense when used in the **command** mode. Statements available in the **command** mode are covered in "Command Mode Statements" on Page 17-27 and "Statements Valid in the Command or Program Mode" on Page 17-33.



## BASIC STATEMENTS

A program is composed of one or more lines or "statements" instructing BASIC to solve a problem. Each program line begins with a line number identifying the line and its statement. The line number indicates the desired order of statement execution. Each statement starts with an English word specifying the operation to be performed. Single statements are terminated with the return key. Multiple statements are separated by a colon (:) with the last statement terminated by a return (a non-printing character). A DATA statement cannot share a line with other statements. (See Page 17-54.)

### Line Numbers

An integer number begins each line in a BASIC program. BASIC executes the program statements in numerical sequence, regardless of the input order. Statement numbers must lie in the range of 1 to 65,535. It is good programming practice to number lines in increments of 5 or 10 to allow insertion of forgotten or additional statements when de-bugging the program.

The length of a BASIC statement must not exceed one line. There is no method to continue a statement to a following line. However, multiple statements may be written on a single line. In this situation each statement is separated by a colon. For example:

```
10 PRINT "VALUES", A, A+1 is a single line print statement, whereas
10 LET A=12: PRINT A, A+1, A+2 is a line containing two statements, LET and PRINT.
```

Virtually all statements can be used anywhere in a multiple statement line. There are, however, a few exceptions. They are noted in the discussion of each statement. NOTE: Only the first statement on a line can have a line number. Program control cannot be transferred to a statement **within** a line, but only to the beginning of a line.

### Statement Types

BENTON HARBOR BASIC supports three different types of statements. First, there are statements valid only in the command mode. These statements are used for loading programs, erasing memory, and other such functions directing BASIC's activities. Second, there are statements valid as both commands or within a program. Third, there are statements valid only within a program. These statements may not be used in the command mode. Most statements fall into the second category. This means they can appear within a program or be typed directly in the command mode and immediately executed.



As noted earlier, some statements valid in both modes may not be meaningful in both modes.

BASIC is designed to allow maximum versatility in its structure. Thus, almost everywhere that BASIC requires a number or a string, BASIC allows a numeric or a string **expression**. For example, you can construct a computed GOTO by simply computing a value for a variable, X. The statement

GOTO X

then redirects the program to the computed line number.

The following three sections are organized as command mode statements, command and program mode statements, and program mode statements. They can be found, respectively in: "Command Mode Statements" (below), "Statements Valid in the Command or Program Mode" (Page 17-33), and "Program Only Statements" (Page 17-58).

To simplify some practical descriptions in these sections and those following, the notation below is used to describe allowed expressions:

1. "iexp" indicates an integer expression, an expression lying in the range of 0 to 65535. The fractional part of any integer expression is discarded when the integer is formed.
2. "nexp" indicates a numeric expression. This may be an integer, decimal, or exponential expression with up to 6 decimal places.
3. "sexp" indicates a string expression. String expressions are limited to a maximum of 255 printing ASCII characters. String expressions are limited to *Extended BASIC*.
4. "sep" indicates a separator. Separators such as the comma and the semi-colon are used to delineate certain portions of BASIC statements.
5. "[ ]" brackets indicate optional portions of a statement, depending on the exact function desired.
6. "var" indicates a variable. This may be a numeric or string variable, depending upon the example.
7. "name" indicates a string used to identify a date, a program, or a language record.

## Command Mode Statements

The command mode statements cannot be used within a program. For example, the RUN statement cannot be used within a program to make it self-starting. Any attempt to incorporate one of these statements within a program generates a USE error message.

### BUILD

*This statement is used to insert or replace many program lines. The form of the BUILD statement is:*

```
BUILD iexp1, iexp2
```

*When BUILD is executed, the initial line number iexp1 is displayed on the terminal. Any text entered after the new line number is displayed becomes the new line, replacing any pre-existing line. Once the line is completed by a carriage return, the next line number is displayed. NOTE: If a null entry is given (a carriage return typed directly after the line number is displayed), the line whose number is displayed is eliminated if it existed.*

*BUILD is illustrated in the following example. CONTROL-C terminates BUILD.*

```
*BUILD 100,10
100 PRINT "LINE 100"
110 PRINT "LINE 110"
120 PRINT "LINE 120"
130 < CNTRL-C > (Control-C typed here)
*LIST
100 PRINT "LINE 100"
110 PRINT "LINE 110"
120 PRINT "LINE 120"
*
```

### CONTINUE

CONTINUE begins or resumes the execution of a BASIC program. CONTINUE has the unique feature of not affecting any existing variable values, nor does it affect the GOSUB or FOR stack. CONTINUE is normally used to resume execution after an error the program or after a CONTROL-C stops the program. CONTINUE may be used to enter a program or a specific line (in conjunction with a GOTO). CONTINUE is unlike RUN, which resets all variables, stacks, etc.. The form of the CONTINUE statement is:

```
CONTINUE
```

In the following example, CONTINUE starts the program at a specific line number.

```
*GOTO 100
*CONTINUE (start execution at line 100)
```

CONTINUE is also useful for entering a program with a variable or variables set at particular values. For example:

```
*A = 23.5 (Program continues execution at Line 230
 with variable A set to the value 23.5,
*GOTO 230 regardless of previous program effects on A.)
*CONTINUE
```

## DELETE

*The DELETE statement is used to remove several lines from the BASIC source program. The form of the DELETE statement is:*

```
DELETE iexp1, iexp2,
```

*The lines between and including iexp1 and iexp2 are deleted.*

*A syntax error is flagged if "iexp1" is greater than "iexp2." Normally, DELETE is used to eliminate a number of lines of text. The SCRATCH command is used to eliminate all text. A RETURN typed directly after a line number eliminates that line. This technique is used to eliminate a single line.*

## DUMP

The DUMP statement saves the current program text on the mass storage media connected to the load/dump port. This is usually paper tape or cassette. The current program is saved; however, no variables are saved. The specific program name is written with the data so the user may reload the program by the specified name. The form of the DUMP statement is:

```
*DUMP "name"
```

Make the tape drive ready before entering the DUMP statement. BASIC starts the drive, writes the data, and stops the drive. The CONTROL-C can be used to abort the DUMP while in progress. However, if a DUMP is aborted, an incomplete file exists on the tape.



The string "name" may be up to 80 ASCII characters. The normal string ASCII characters are permitted. An example of a DUMP is:

```
*DUMP "STARTREK VER 1.0 03/11/77"
```

This statement dumps the program Startrek version 1.0 dated the 11th of March 1977.

## LOAD

The LOAD statement discards the current program in memory. A specified program is loaded from the mass storage device connected to the load/dump port. The form of the LOAD statement is:

```
LOAD "name"
```

The string "name" may consist of up to 80 ASCII characters. The normal string ASCII characters are permitted. BASIC scans the mass storage device until it finds a program whose name matches the specified string. Before destroying the stored information, the user is asked "SURE?." A "Y" reply causes LOAD to proceed. Any other response cancels LOAD.

NOTE: If the name on the mass storage device is longer than the specified name, a match on the supplied characters in the string "name" is valid. Thus, a program may be dumped with extra information entered in the name such as program version number and data. The program can then be loaded without it. For example:

```
*DUMP "STARTREK VER 1.0 03/11/77"
```

This program, Startrek version 1.0 dated 11 March 1977, may be loaded by

```
*LOAD "STARTREK"
SURE? Y
```

A match is found between the first eight characters of the DUMP string "STAR-TREK" and the eight characters of the load command "STARTREK." If a null is used as the load string, the next program on the tape is loaded. Therefore, the statement

```
*LOAD ""
```

loads the next BASIC program appearing on the mass storage.

Mass storage media should be made ready before LOAD is executed. BASIC starts and stops the mass storage device. CONTROL-C may be used to abort the load part way through. Use a SCRATCH command to clear the results of an aborted load.

During a load, either one of the following two error messages may be generated:

SEQ ERR and  
CHKSUM ERR.

A sequence error (SEQ ERR) is generated if the file records are not in sequence. For example, if two consecutive label records are read an error is generated, as a BASIC file consists of a label file followed by a data file. The form of the sequence error is

SEQ ERR

Type a blank after the SEQ ERR message. This will clear the error. The entire file must be reread.

A checksum error (CHKSUM ERR) is generated if the computed CRC for the record in question does not match the CRC included in the record. The form of the checksum error message is

CHKSUM ERR    IGNORE?

A Y in response to the question "ignore" aborts the error message and the next consecutive record is read. Do not ignore the checksum error unless there is no other way to recover the data. If a checksum error is flagged, the chances are very good that the data in the designated record is faulty. If you attempt to use such bad data, it may cause BASIC to crash.

## RUN

A prepared program may be executed using the RUN statement. The program is executed starting at the lowest numbered statement. All variables and stacks are cleared (set to zero) before program execution starts.

The form of the RUN statement is:

\*RUN

After program completion, BASIC prompts the user with an asterisk (\*) in the left margin, indicating that it is ready for additional command statements. If the program should contain errors, an error message is printed that indicates the error and the line number containing the error, and program execution is terminated. Again, a prompt is given. The program must now be edited to correct the error and rerun. This process is continued until the program runs properly without producing any error messages. See "Errors" (Page 17-75) for a discussion of error messages.

Occasionally a program contains an error that causes it to enter an unending loop. In this case, the program never terminates. The user may regain control of the program by typing CONTROL-C (CNTRL-C). This aborts the program and returns control to the user. Storage is not altered in this process. CONTINUE resumes program execution. RUN clears the storage and restarts program execution.

## SCRATCH

SCRATCH clears all current storage areas used by BASIC. This deletes any commands, programs, data, strings, or symbols currently stored by BASIC.

SCRATCH should be used for entering a new program from the terminal keyboard to ensure that old program lines are not mixed with new program lines. It also assures a clear symbol table. The form of the SCRATCH statement is:

\*SCRATCH

Before destroying stored information, the user is asked "SURE?" A "Y" reply causes SCRATCH to proceed. Any other response cancels SCRATCH. For example:

|          |                                   |
|----------|-----------------------------------|
| *SCRATCH | (Scratch statement entered.)      |
| SURE? Y  | (Are you sure, answer Y (YES,))   |
| *        | (BASIC is ready for a new entry.) |

## VERIFY

The VERIFY statement permits you to check a file placed on mass storage without affecting the current program. The VERIFY command responds by indicating the name of the file found, and if the file is correct. A form of the VERIFY command is:

\*VERIFY "name"



The string "name" can be the name of the record the user desires to verify or it may be a null (""); in which case, BASIC verifies the first record encountered. For example,

```
*VERIFY "STARTREK"
FOUND STARTREK VER 1.0 03/11/77
FILE OK
*
```

In the above example, the file containing the Startrek dump is verified. Note, that the name of the file is printed immediately as soon as the label record is encountered. The FILE OK message is printed after the data record is read and verified. VERIFY performs a checksum on the contents of all data in the file. Using the VERIFY command does not destroy any program data in memory.

During a VERIFY, one of two error messages may be generated. They are:

SEQ ERR and  
CHKSUM ERR.

A sequence error (SEQ ERR) is generated if the file records are not in sequence. For example, if two consecutive label records are read an error is generated, as a BASIC file consists of a label file followed by a data file. The form of the sequence error is

SEQ ERR

Type a blank after the SEQ ERR message. This will clear the error. The entire file must be reread.

A checksum error (CHKSUM ERR) is generated if the computed CRC for the record in question does not match the checksum included in the record. The form of the CRC error message is

CHECKSUM ERR - IGNORE?

A Y in response to the question ignore aborts the error message and the record is considered valid. Do not ignore the checksum error unless there is no other way to recover the data. If a checksum error is flagged, the chances are very good that the data in the designated record is faulty.

The command VERIFY is not available if BASIC is patched to use an ASR console terminal as the load dump device. See "Product Installation" on Page 0-19 of the "Introduction" to this Software Reference Manual.

## Statements Valid in the Command or Program Mode

The statements in this section may be used in either the command or the program mode. A few of them have only subtle uses in one mode or the other. Because they may be used in both modes, they are listed in this section.

### CLEAR

CLEAR sets the contents of all variables, arrays, string buffers, and stacks to zero. The program itself is not affected. The command is generally used before a program is rerun to insure a fresh start if the program is started with a command other than RUN. The form of the CLEAR statement is:

```
*10 CLEAR (BASIC)
10 CLEAR varname (EXTENDED BASIC)
```

All variables, arrays, string buffers, etc., are cleared before program is executed by RUN. Therefore, a clear statement is not required. However, a program terminated prior to execution (by a STOP command or an error) does not set these variables, etc., to zero. They are left with the last value assigned. *If the variable name (varname) is specified, the CLEAR command clears the named variable, array, or DEF FN (user defined function).*

*Note that the memory space used by string variables and arrays is not freed when CLEAR varname is used. String values should be set to null (for example, A\$ = "") before clearing so the string space can be recovered.*

For example:

```
CLEAR A Clears variable A
CLEAR A$ Clears the string variable A$
CLEAR A(Clears the dimensioned variable A(
```

If a section of the program is to be rerun after appropriate editing, the variables, arrays, dimensions, etc., should be reinitialized. You can accomplish this by using the CLEAR statement in the command mode.

**CNTRL (CONTROL)**

*CONTROL is a multi-purpose command used to set various options and flags. The form of the CONTROL statement is:*

```
CNTRL iexp1, iexp2
```

*The various CNTRL options are:*

|       | iexp1 | iexp2 |
|-------|-------|-------|
| CNTRL | 0,    | nnn   |
| CNTRL | 1,    | n     |
| CNTRL | 2,    | n     |
| CNTRL | 3,    | n     |
| CNTRL | 4,    | n     |

**CNTRL 0**

*The CNTRL 0, nnn command sets up a GOSUB routine to process CONTROL-B characters. The line number of the routine is specified as "iexp2." When a CONTROL-B is entered from the terminal program, control is passed to the specified statement (beginning at the line iexp2) via a GOSUB linkage, after the statement being executed is completed. For example:*

```
10 CNTRL 0,500
20 FOR A=1 TO 9
30 PRINT A,A*A,A*A*A
40 NEXT A
50 END
500 PRINT "THAT TICKLES"
510 RETURN
```

```
*RUN
```

|   |   |                      |
|---|---|----------------------|
| 1 | 1 | 1                    |
| 2 | 4 | 8                    |
| 3 | 9 | 27 (CONTROL-B typed) |

```
THAT TICKLES
```

|   |    |                      |
|---|----|----------------------|
| 4 | 16 | 64 (CONTROL-B typed) |
|---|----|----------------------|

```
THAT TICKLES
```

|   |    |     |
|---|----|-----|
| 5 | 25 | 125 |
| 6 | 36 | 216 |
| 7 | 49 | 343 |
| 8 | 64 | 512 |
| 9 | 81 | 729 |

```
END AT LINE 50
```

```
*
```



During the execution of the program containing these three statements, a **CONTROL-B** from the keyboard momentarily interrupts the regular execution of the program. The program completes the line in progress and then enters the subroutine at line 500 printing the string

THAT TICKLES

It then moves to the next statement, a **RETURN**. This causes the program to continue with normal program execution. **NOTE:** The **CNTRL 0, nnn** must be executed before it is operational.

### CNTRL 1

The **CNTRL 1, n** command sets the number of digits permitted before the exponential notation is used. Normal mode  $M = 6$ . For example:

**CNTRL 1,2** (Numbers  $\geq 100$  are to be in exponential format.)

```
*PRINT 101
1.01000E+02
```

### CNTRL 2

The **CNTRL 2, n** command controls the H8 front panel LED display mode. The control functions are:

**CNTRL 2,0** Turn display off (Normal mode).

**CNTRL 2,1** Turn display on without update. (For writing into a display. See the example under "The SEG Function, SEG (NARG)" on Page 17-65).

**CNTRL 2,2** Turn display on with update (to monitor a register or memory location).

### CNTRL 3

The **CNTRL 3, n** command controls the size of a print zone. This is normally 14. However, **CNTRL 3, n** can change the number of spaces in a print zone.

```
*
*CNTRL 3,5
*PRINT 1,2,3,4,3,2,1,0
 1 2 3 2 3 4 3 2 1 0
```

**CNTRL 4**

The **CNTRL 4,n** command turns the hardware clock on and off. **CNTRL 4,0** turns the clock off and **CNTRL 4,1** turns it on. Once the clock has been turned off, clock dependent functions such as **PAUSE iexp**, and **PAD(** cannot be used. Turning the clock off increases execution speed approximately 11%.

**NOTE:** The **CNTRL 1** through **CNTRL 4** commands permanently reconfigure loaded **BASIC**. A new program does not clear them to the original state. You can reset them to their original state by using the appropriate form of the **Control** statement in the **Command** mode.

**DIM (DIMENSION)**

The **DIMENSION** statement explicitly defines the maximum dimensions of array variables. A single dimension array is often called a vector. The form of the **DIMENSION** statement is:

```
*DIM varname (iexp1[, ,iexpn]) [,varname2 (.)]
```

The expressions “iexp1” through “iexpn” are integer expressions specifying the bounds of each dimension. Dimensions are 0 to “expn.” So, for example, the statement:

```
DIM A(5,5)
```

reserves an array 6×6 or 36 values. If the dimensioned variable is numeric, the values are preset to zero. If the dimensioned variable is a string, all the values are preset to a null string.

You may declare several variables in one **DIMENSION** statement by separating them with commas. For example:

```
*DIM A6(3,2), B(5,5), C3(10,10)
```

dimensions the following arrays

| <u>VARIABLE</u> | <u>SIZE</u> |              |
|-----------------|-------------|--------------|
| A6              | 4 by 3      | 12 elements  |
| B               | 6 by 6      | 30 elements  |
| C3              | 11 by 11    | 121 elements |

You can place a DIMENSION statement anywhere in a multiple statement line and it can appear anywhere in the program. However, an array can only be dimensioned once in a program unless it is cleared. DIMENSION statements must be executed before the first reference to the array, although good programming practices place all DIMENSION statements in a group among the first statements of a program. This allows them to be easily identified and changed if alterations are required later. The following example demonstrates the use of the DIMENSION statement with subscripted variables and a two-level FOR statement.

```
*LIST
10 REM DIMENSION DEMO PROGRAM
20 DIM A(5,10)
30 FOR B=0 TO 5
40 LET A(B,0)=B
50 FOR C=0 TO 10
60 LET A(0,C)=C
70 PRINT A(B,C);
80 NEXT C:PRINT :NEXT B
90 END
```

```
*RUN
0 1 2 3 4 5 6 7 8 9 10
1 0 0 0 0 0 0 0 0 0 0
2 0 0 0 0 0 0 0 0 0 0
3 0 0 0 0 0 0 0 0 0 0
4 0 0 0 0 0 0 0 0 0 0
5 0 0 0 0 0 0 0 0 0 0
```

END AT LINE 90

\*

## FOR AND NEXT

FOR and NEXT statements define the beginning and end of a program loop. A program loop is a set of repeated instructions. Each time they are repeated they modify a variable in some way until a predetermined condition is reached, causing the program to exit from the loop. The FOR NEXT statement is of the form:

```
FOR var = nexp1 to nexp2 [STEP nexp3]
NEXT VAR
```



When BASIC encounters the FOR statement, the expressions nexp1, nexp2 and nexp3 (if present) are evaluated. The variable "var" may be a scaler numeric variable, or it may be an element of a numeric array. It is assigned a value of "nexp1." For example:

```
*FOR A=2 TO 20 STEP 2:PRINT A::NEXT A
 2 4 6 8 10 12 14 16 18 20
```

causes the program to execute as long as A is less than or equal to 20. Each time the program passes through the loop, the variable A is incremented by 2 (the STEP number). Therefore, this loop is executed a total of 10 times. When incremented to 22, program control passes to the line following the associated NEXT statement. It is important to note that the initial value used for the variable is the value assigned to the variable expression when it entered the FOR-NEXT loop. For example:

```
*A=10:FOR A=2 TO 20 STEP 2:PRINT A::NEXT A
 2 4 6 8 10 12 14 16 18 20
*
```

Prior to execution, the variable A is assigned the value 10. The program passes through the loop 10 times. A is reset to 2 and then increments from 2 to 20.

If "nexp2"  $\geq$  0, and the initial value of var  $\geq$  "nexp2," the loop terminates. For example, the program:

```
*LIST
10 FOR J=2 TO 18 STEP 4
20 J=18
30 PRINT J::NEXT J
40 END
```

```
*RUN
18
END AT LINE 40
*
```

is only executed once, since the value of J = 18 is reached on the first pass, satisfying the termination condition.

A loop created by the statement:

```
*FOR A=20 TO 2 STEP 2:PRINT A;:NEXT A
20
*
```

is executed only once, as the initial value exceeds the terminal value. However, if this example is modified to read:

```
*FOR A=20 TO 2 STEP -2:PRINT A;:NEXT A
20 18 16 14 12 10 8 6 4 2
*
```

the negative step allows normal operation.

In summary, for positive STEP values, the loop is executed until the variable (var) is greater than the final assigned value (nexp2). For negative STEP values, the loop is executed until the variable (var) is less than the final assigned value (nexp2).

If the loop does not terminate, execution is transferred to the statement following the FOR statement. Therefore, a series of statements may be executed using the incremented value of the variable. If the loop does terminate, execution is transferred to the statement following NEXT.

The expressions in the FOR statement can be any acceptable BASIC numeric expressions.

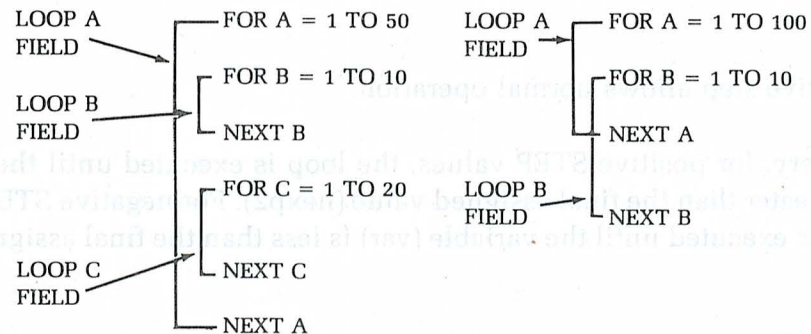
If the STEP expression and the word STEP are omitted from the FOR statement, a step of +1 is the default value. Since +1 is an extremely common step value, the STEP portion of the statement is frequently omitted. For example:

```
*FOR A=2 TO 10:PRINT A;:NEXT A
2 3 4 5 6 7 8 9 10
*
```

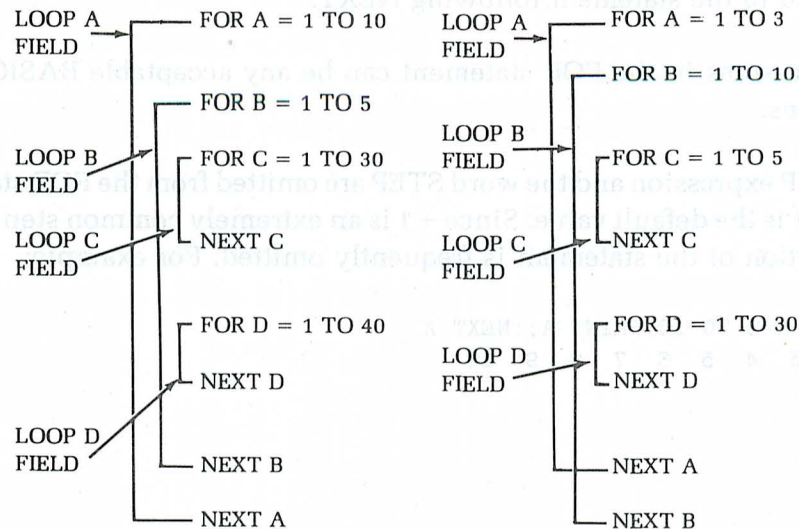
Nesting is a technique frequently used in programming. It consists of placing one or more loops completely inside another loop. The field or operating range of the loop (the lines from the FOR statement to the corresponding NEXT statement) must not cross the field of another loop. The following two examples show legal and illegal nesting of FOR NEXT loops.

LEGAL NESTINGILLEGAL NESTING

## Two-Level Nesting



## Three-Level Nesting



Note that both columns of nesting illustrations are shown in two-level and three-level forms. However, right-hand columns are not truly nesting but a crossover of FOR and NEXT loops (fields), and therefore are illegal. Also note that each of these examples uses the implied STEP value of 1.



The depth of nesting depends upon the amount of memory space available in Extended BASIC. BASIC limits FOR loops to 5 levels. Exceeding 5 levels generates an overflow error.

It is possible to exit from a FOR NEXT loop without reaching the variable termination value. This can be done using a conditional transfer such as an IF statement within the loop. However, control can only be transferred into a loop if the loop is left during prior program execution without being completed. This ensures the assignment of values to the termination and step variables.

Both FOR and NEXT statements can appear anywhere on a multiple statement line.

The NEXT statement does not require the variable. If the variable is not given, BASIC will NEXT the innermost FOR loop.

### FREE (EXTENDED BASIC ONLY)

The FREE statement displays the amount of memory used by EX. B.H. BASIC and any program material. It also displays the total amount of free space left, which is dependant on the amount of memory in the computer and the program size. This command is particularly valuable when you are gauging the size of the program's data structure and establishing limits on a DIMENSION command. The FREE command also indicates the cause of memory overflow errors. The form of the FREE statement is:

\*FREE

The form of the printout is:

|             |                                                     |
|-------------|-----------------------------------------------------|
| TEXT = nnnn | (Bytes used by program text.)                       |
| SYMB = nnnn | (Bytes used by variables and arrays.)               |
| FORL = nnnn | (Bytes used by FOR loops.)                          |
| GSUB = nnnn | (Bytes used by GOSUBs.)                             |
| STRN = nnnn | (Bytes used by STRING.)                             |
| WORK = nnnn | (Bytes used by expression and function evaluation.) |
| FREE = nnnn | (Total number of free bytes.)                       |

For example, running the program

```
*10 GOSUB 10
```

BASIC soon returns a memory overflow error. Executing FREE shows the user a very large GOSUB table. This, and the statement provided in the error message, enables one to determine the program is in a GOSUB loop.

```
*FREE
TEXT = 9
SYMB = 0
FORL = 0
GSUB = 0
WORK = 0
STRN = 0
FREE = 7248
*10GOSUB 10
*RUN
```

```
! ERROR - MEM OVR AT LINE 10
```

```
*FREE
TEXT = 9
SYMB = 0
FORL = 0
GSUB = 7232
WORK = 0
STRN = 0
FREE = 16
*
```

## GOSUB AND RETURN

A subroutine is a section of program performing some operation required one or more times during program execution. Complicated operations on a volume of data, mathematical evaluations too complex for user defined functions, or a previously written routine are all examples of processes best performed by a subroutine.

More than one subroutine is allowed in a single program. Good programming practices dictate that subroutines should be placed one after another at the end of the program in line number sequence. A useful practice is to assign distinctive line number groups to subroutines.

For example, a main program uses line numbers through 300. The 400 block is assigned to subroutine #1 and the 500 block is assigned to subroutine #2. Thus, any errors, program modifications, etc., involving the subroutine are easily identified.

Subroutines are normally placed at the end of a program, but before data statements if there are any.

Program execution begins and continues until a GOSUB statement is encountered. The form of the GOSUB statement is:

```
*GOSUB iexp
```

where iexp is the first line in the subroutine. Once GOSUB is executed, program control transfers to the first line of the subroutine and the subroutine is executed. For example:

```
60 GOSUB 500
```

in this example, control (the sequence of program execution) is transferred to line 500 in the program after line 60 is executed. The first line in the subroutine may often be a remark to identify the subroutine, or it may be any executable statement.

Once program control is transferred to a subroutine, program execution continues in the normal line-by-line manner until a RETURN statement is encountered. The RETURN statement is of the form:

```
RETURN
```

RETURN causes the program control to return to the statement **following** the original GOSUB statement. A subroutine must always be terminated by a RETURN.

Before BASIC transfers control to a subroutine, the next sequential statement to be processed after the GOSUB statement is internally recorded. The RETURN statement draws on this stored information to restart normal program execution. Using this technique, BASIC always knows where to transfer control, no matter how many times subroutines are called.

Subroutines can be nested in the same manner that FOR NEXT statements can be nested. That is, one subroutine can call another subroutine, and if necessary, that subroutine may call a third subroutine, etc. If, during execution of the subroutine a RETURN is encountered, control is returned to the line following the GOSUB calling the subroutine. Therefore, a subroutine can call another subroutine, even itself. Subroutines can be entered at any point and can have more than one RETURN. Multiple RETURN statements are often necessary when a subroutine contains conditional statements imbedded in it, which cause different subroutine completions dependent on the program data.



It is possible to transfer to the beginning or to any part of the subroutine. Multiple entry points and returns make the GOSUB statement an extremely versatile tool.

Up to 10 levels of GOSUB nesting are permitted in BASIC. *Extended BASIC permits unlimited GOSUB nesting. However, nesting uses memory and excessive nesting depth will cause an overflow.*

## GOTO

The GOTO statement provides unconditional transfer of program execution to another line in the program. The GOTO statement is of the form:

\*GOTO iexp

When this statement is executed, program control transfers to the line number specified by the integer expression "iexp." For example:

```
10 LET A=1
20 GOTO 40
30 LET A=2
40 PRINT A
50 END
```

\*RUN

1

END AT LINE 50

\*

Program lines in this example are executed in the following order:

10, 20, 40, 50

Line 30 is never executed because the GOTO statement in line 20 unconditionally transfers control to line 40. After the unconditional transfer takes place, normal sequential execution resumes at line 40.

## IF THEN (IF GOTO)

The IF THEN (IF GOTO) conditionally transfers program execution from the normal consecutive order of program lines, depending on the results of a relation test. The forms of the IF statement are:

IF expression  $\left\{ \begin{array}{l} \text{THEN} \\ \text{GOTO} \end{array} \right\} \text{iexp} \quad \text{or}$   
 IF expression THEN statement

The expression frequently consists of two variables combined by the relational operators described in "Relational Operators" (Page 17-18). In the first form, if the result of the expression is true, control passes to the specified line number (iexp). In the second form, control passes to the statement following THEN on the remainder of the line. If the result of the expression is false, control passes to the next line or to a statement separated from the IF THEN statement by a colon (:). The following examples show use of the IF THEN statement.

```
10 READ A
20 B=10
30 IF A=B THEN 50
40 PRINT "A< >B",A:END
50 PRINT "A=B",A
60 DATA 10,5,20
70 END
```

\*RUN

A=B 10

END AT LINE 70

\*CONTINUE

A< >B 5

END AT LINE 40

\*CONTINUE

A< >B

END AT LINE 40

\*

NOTE: The expression can be an arbitrarily complex expression. For example:

IF (A<3) AND NOT (B>C) THEN

## LET

The LET statement assigns a value to a specific variable. The form of the LET statement is:

```
LET var = nexp, or
LET var$ = sexp
```

The variable "var" may be a numeric variable or a string variable "var\$." The expression may be either an arithmetic "nexp" or a string expression "sexp" (*Extended BASIC*). However, all items in a statement must be either numeric or string, they cannot be mixed. If they are mixed, a type conflict error is flagged. NOTE: Unlike standard BASIC, multiple assignments are not permitted. For example,

```
LET A=B=3
```

causes A to be set to 65,535 (true) if B is equal to 3, or it causes A to be set to 0 (false) if B is not equal to 3. It does not cause both A and B to be set to 3.

You may omit the key word LET if you prefer. For example, the following two statements produce identical results.

```
10 LET A = 6
AND
10 A = 6
```

The LET statement is often referred to as an assignment statement. In this context, the meaning of the equal (=) symbol should be understood as it is used in BASIC. In ordinary algebra, the formula  $Y = Y + 1$  is meaningless. However, in BASIC the equal sign denotes replacement rather than equality. Thus, the formula  $Y = Y + 1$  is translated as add 1 to the current value of Y and store the new result at the location indicated by the variable Y.

Any values previously assigned to Y are combined with 1. An expression such as  $D = B + C$  instructs BASIC to add the values assigned to the variables B and C and store the resultant value at the location indicated by the variable D. The variable D is not evaluated in terms of previously assigned values, but only in terms of B and C. Therefore, if previous assignments gave D a different value, the prior value is lost when this statement is executed.



## LIST

This command lists the program on the console terminal for reviewing, editing, etc. The form of the list command is:

```
LIST [iexp] B.H. BASIC
LIST [iexp1], [iexp2] EX. B.H. BASIC
```

Line numners are indicated by the optional integer expressions. If no line numbers are specified, the entire program is listed. *If a single line number ("iexp 1") is specified, EX. B.H. BASIC lists that single line.* BASIC lists the indicated line and the balance of the program lines. You can use a CONTROL-O or CONTROL-C to abort the listing *If both of the optional line numbers are specified, separated by a comma (,), all lines within the range of iexp 1 to iexp 2 are listed.* You can abort a listing by using the control characters. Refer to "Editing Commands" (Page 17-71) or to "Appendix B" (Page 17-83) for a complete explanation of these functions.

The following are examples of the LIST command.

```
10 LET A=5:LET B=6
20 PRINT A,B,A+B,
30 LET C=A/B
40 PRINT C
50 END
*RUN
5 6 11 .833333
```

END AT LINE 50

\*LIST

```
10 LET A=5:LET B=6
20 PRINT A,B,A+B,
30 LET C=A/B
40 PRINT C
50 END
*LIST 20
```

```
20 PRINT A,B,A+B,
*LIST 20,40
```

```
20 PRINT A,B,A+B,
30 LET C=A/B
40 PRINT C
*
```

} EX. B.H. BASIC only

## ON ... GOSUB

The ON ... GOSUB statement allows you to program a computed GOSUB. When you use the ON ... GOSUB statement, use a RETURN at the end of the sub-routine to return program control to the statement **following** the ON ... GOSUB statement. The form of the ON ... GOSUB statement is:

```
ON iexp1 GOSUB iexp2,, iexpn
```

When it is processing an ON ... GOSUB statement, BASIC evaluates the expression "iexp1" and uses the result as an index to the list of statement numbers iexp2 thru iexpn. If the expression "iexp1" evaluates to 1, for example, control is passed to the line number given by the expression "iexp2." If the expression "iexp1" evaluates to 3, for example, control is passed to line number given by the expression "iexp4." If the expression "iexp1" evaluates to 0, or to an index greater than the number of statement numbers listed, control is passed to the next program statement.

## ON ... GOTO

The ON ... GOTO statement allows you to perform a computed GOTO. The form of the ON ... GOTO statement is:

```
ON iexp1 GOTO iexp2,, iexpn
```

When it is processing an ON ... GOTO statement, BASIC evaluates the expression "iexp1" and uses the result as an index to the list of statement numbers iexp2 thru iexpn. If the expression "iexp1" evaluates to 1, for example, control is passed to the line number given by the expression "iexp2." If the expression "iexp1" evaluates to 3, for example, control is passed to line number given by the expression "iexp4." If the expression "iexp1" evaluates to 0, or to an index greater than the number of statement numbers listed, control is passed to the next program statement.

## OUT

The OUT statement is used to output binary numbers to an output port. The form of the OUT statement is:

```
OUT iexp1, iexp2
```

The expression "iexp1" is used as the port address, and "iexp2" is the value to be placed at that port. Both iexp1 and iexp2 are decimal numbers. The low-order 8-bits generated by the decimal numbers in iexp1 or iexp2 are used. If you wish to write iexp1 and iexp2 in octal notation for ease in conversion to the actual binary values, write a subroutine or function to perform octal to decimal conversion.

## PAUSE

The PAUSE statement causes BASIC to delay before executing the next statement. There are two forms of PAUSE. In BASIC the form of the PAUSE statement is:

```
PAUSE
```

Once the PAUSE statement is executed, no further statements are executed until you type a console terminal character. You can terminate PAUSE by typing any key, and this will not cause the character, to be echoed, but it is good practice to consistently use one character such as space to terminate PAUSE. *In Extended BASIC the form of the PAUSE statement is:*

```
PAUSE [iexp]
```

*You can also terminate PAUSE by specifying an optional time duration with iexp. If iexp is specified, PAUSE waits 2 times iexp milliseconds. After 2 times iexp milliseconds pass, normal program execution continues.*

The PAUSE statement is particularly useful when you are viewing long outputs on a CRT display. You can insert a PAUSE at appropriate points in the program, allowing you to view the information on the CRT before continuing execution.

## POKE

The POKE statement is used to write values into an assigned H8 memory location. The form of the POKE statement is:

```
POKE iexp1, iexp2
```

The low-order 8-bits of iexp2 are inserted into memory location iexp1. **NOTE:** iexp1 and iexp2 must be given as decimal numbers. If you wish to use octal numbers for ease in referencing to binary notation, you must use a separate octal to decimal subroutine or function to generate these numbers.



**CAUTION:** You can damage BASIC when using the POKE statement, causing a failure which could result in loss of program material and/or require reloading BASIC itself. The POKE statement should be confined to areas of memory, not used by the BASIC interpreter.

## PORT

The PORT statement is used to direct the result of a print statement to an I/O port other than the console terminal port (372<sub>8</sub> or 250<sub>10</sub>). You can also use it to direct the console terminal operations to another port. One of the primary uses of the PORT statement is directing data printing to a printer. The form of the PORT statement is:

```
PORT iexp
```

The expression iexp is the desired port number (in decimal). If iexp is positive, the print function is transferred to the indicated port number while the statement (in the command mode) or the program (in the program mode) is being executed. After execution is complete, the printing function returns to the console terminal. If iexp is negative, the keyboard functions of the console terminal are transferred to the desired port, along with the printing functions.

The console terminal is then permanently transferred to the selected port and an additional PORT instruction must be issued at this new console terminal to return operation to port 250. Recovery from accidental assignment to a non-existent port is accomplished by an RST0 followed by a warm start (PC = 040 103).

## PRINT

The PRINT statement is used to output **data** to the console terminal. The form of the PRINT statement is:

```
PRINT [nexp1 sep1 . . . nexpn(sepn)]
```

The expressions and separators contained within the brackets are optional. When used without these optional expressions and separators, the simple PRINT statement outputs a blank line followed by a carriage-return line feed.

### Printing Variables

The PRINT statement can be used to evaluate expressions and to simultaneously print their results, or to simply print the results of a previously evaluated

expression or evaluations. Any expression contained in the PRINT statement is evaluated before the result is printed. For example:

```
10 A=4:B=6:C=5+A
20 PRINT
30 PRINT A+B+C
40 END
*RUN
```

```
19
```

```
END AT LINE 40
```

```
*
```

All numbers are printed with a preceding and following blank. You can use PRINT statements anywhere in a multiple statement line. NOTE: The terminal performs a carriage-return line feed at the end of each PRINT statement unless you use the separators described in "Use of the , and ;" (Page 17-52). Thus, in the previous example, the first PRINT statement outputs a carriage-return line feed and the second print statement outputs the number 19 followed by a carriage-return line feed.

## Printing Strings

The PRINT statement can be used to print a message (a string of characters). The string may be alone or it may be used together with the evaluation and printing of a numeric value. Characters to be printed are designated by enclosing them in quotation marks ("). For example:

```
10 PRINT "THIS IS A HEATH H8"
*RUN
THIS IS A HEATH H8

END AT LINE 65535
*
```

The string contained in a PRINT statement may be used to document the variable being printed. For example:

```
10 LET A=5:LET B=10
20 PRINT "A + B",A+B
30 END
*RUN
A + B 15

END AT LINE 30
*
```

When a character string is printed, only the characters between the quotes appear. No leading or trailing blanks are added as they are when a numeric value is printed. Leading and trailing blanks can be added within the quotation marks.

### Use of the , And ;

The console terminal is normally initialized with 80 columns divided into five zones. (See CNTRL 3, n for exception.) Each zone, therefore, consists of 14 spaces. When an expression in the PRINT statement is followed by a comma (,) the next value to be printed appears in the next available print zone. For example:

```
10 A=5.55555:B=2
20 PRINT A,B,A+B,A*B,A-B,B-A
30 END
*RUN
5.55554 2 7.55554 11.1111 3.55554
-3.55554

END AT LINE 30
*
```

NOTE: The sixth element in the PRINT list is the first entry on a new line, as the five print zones of a 72-character line were used.

Using two commas together in a PRINT statement causes a print zone to be skipped. For example:

```
10 A=5.55555:B=2
20 PRINT @,B,A+B,,A*B,A-B,B-A
30 END
*RUN
5.55554 2 7.55554 11.1111
2.55554 -3.55554

END AT LINE 30
*
```



If the last expression in a PRINT statement is followed by a comma, no carriage-return line feed is given when the last variable is printed. The next value printed (by a later PRINT statement) appears in the next available print zone. For example:

```

10 LET A=1:LET B=2:LET C=3
20 PRINT A,
30 PRINT B
40 PRINT C
50 END
*RUN
1 2
3

END AT LINE 50
*
```

At certain times, it is desirable to use more than the designated five print zones. If such tighter packing of the numeric values is desired, a semicolon (;) is inserted in place of the comma. A semicolon does not move the next output to the next PRINT zone, but simply prints the next variable, including its leading and trailing blank. For example:

```

10 LET A=1:LET B=2:LET C=3
20 PRINT A;B;C
30 PRINT A+1;B+1
40 PRINT C+1
50 END
*RUN
1 2 3
2 3
4

END AT LINE 50
*
```

**NOTE:** If either a comma or a semicolon is the final character in a PRINT statement, no final carriage-return line feed is printed.

## READ AND DATA

The READ and DATA statements are used in conjunction with each other to enter data into an executing program. One statement is never used without the other. The form of the statements are:

```
READ var1, . . . , varn
DATA exp1, . . . , expn
```

The READ statement assigns the values listed in the DATA statement to the specified variables var1 through varn. The items in the variable list may be simple variable names, arrays, or *string variable names*. Each one is separated by a comma. For example:

```
5 DIM A (2,3)
10 READ C,B$,A (1,2)
20 DATA 12,"THIS IS SIX",56
30 PRINT C,B$,A (1,2)
*RUN
12 THIS IS SIX 56

END AT LINE 65535
*
```

Because data must be read before it can be used in the program, READ statements generally occur in the beginning of a program. You may, however, place a READ statement anywhere in a multiple statement line. The type of expression in the DATA statement must match the type of corresponding variable in the READ statement. When the DATA statement is exhausted, BASIC finds the next sequential DATA statement in the program. NOTE: BASIC does not automatically go to the next DATA statement for every READ statement. Therefore, one DATA statement may supply values for several READ statements if DATA statement contains more expressions than the READ statement has variables.

DATA statements may contain arbitrarily complex expressions to represent the data values. Each value expression is separated from other value expressions by a comma. A field in the DATA statement may be left null by means of two adjacent commas. This causes the associated variable to retain its old value. For example:

```
10 A=1:B=1:C=1
20 READ A,B,C
30 PRINT A,B,C
40 DATA 3,,4
50 END
*RUN
3 1 4

END AT LINE 50
*
```

If a DATA statement appears on a line, it must be the only statement on the line. DATA statements may not follow any other statement on the line. Other statements should not follow DATA statements.

DATA statements do not have to be executed to be used. That is, they may be the last statement in a program, and be used by a READ statement executed earlier in the program. However, if DATA statements appear in a program in such a place that they are executed (there are executable statements beyond the DATA statement), the executed DATA statement has no effect. Therefore, location of DATA statements is arbitrary as long as the expressions contained within the DATA statements appear in the correct order. However, good programming practice dictates all DATA statements occur near the end of the program. This makes it easy for the programmer to modify the DATA statements when necessary.

If an expression contained in a DATA statement is bad, the illegal character error message is printed. All subsequent READ statements also cause the message. If there is no data available in the data table for the READ statement to use, the no data error message is printed.

If the number of expressions in the data list exceed those required by the program READ statements, they are ignored, and thus not used.

### REM (REMARK)

The REMARK statement lets you insert notes, messages, and other useful information within your program in such a form that it is not executed. The contents of the REMARK statement may give such information as the name and purpose of the program, how the program may be used, how certain portions of the program work, etc.. Although the REMARK statement inserts comments into the program without affecting execution, they do use memory which may be needed in exceptionally long programs.

REMARK statements must be preceded by a line number when used in the program. They may be used anywhere in a multiple statement line. The message itself can contain any printing character on the keyboard and can include blanks. BASIC ignores anything on a line following the letters REM.



**RESTORE**

The RESTORE statement causes the program to reuse data starting at the first DATA statement. It resets the DATA statement pointer to the beginning of the program. The RESTORE statement is of the form:

RESTORE

For example:

```
10 READ A,B,C
20 PRINT A,B,C
30 RESTORE
40 READ D,E,F
50 PRINT D,E,F
60 DATA 1,2,3,4,5,6,7,8
70 END
```

\*RUN

|   |   |   |
|---|---|---|
| 1 | 2 | 3 |
| 1 | 2 | 3 |

END AT LINE 70

\*

This program does not utilize the last five elements of the DATA statement. The RESTORE command resets the DATA statement pointer and the READ D,E,F, statement uses the first three data elements, as does the initial READ statement.

The CLEAR command includes the RESTORE function.

## STEP

The STEP command permits you to step through a program a single line or a few lines at a time. The form of the step command is:

```
STEP iexp
```

where the integer expression iexp indicates the number of lines to be executed before stopping. Execution of the desired lines is indicated by the prompt NXT = nnnn, where nnnn is the next line number to be executed. A STEP 2 is required to execute the first program line. All future single-line executions require a STEP or STEP 1. For example:

```
10 READ A,B,C
20 PRINT A,B,C
30 RESTORE
40 READ D,E,F
50 PRINT D,E,F
60 DATA 1,2,3,4,5,6,7,8
70 END
```

```
*CLEAR
```

```
*STEP 3
```

```
1 2 3
```

```
NXT= 30
```

```
*STEP
```

```
NXT= 40
```

```
*STEP
```

```
NXT= 50
```

```
*STEP
```

```
1 2 3
```

```
NXT= 60
```

```
*STEP 2
```

```
END AT LINE 70
```

```
*
```

## Program Mode Statements

PROGRAM MODE statements are valid only when utilized within a program. If they are entered in the command mode, an illegal use error is flagged.

### DATA

The DATA statement discussed in "Read and Data" (Page 17-54) is a program only statement, although it is used in conjunction with a READ statement, which may be used in either the command or program modes.

### DEF FN

The DEF FN statement defines single line program functions created by the user. The form of the DEF FN statement is:

```
DEF FN varname (arg1 [,arg2, ,argn]) = expr
```

The variable name (varname) must be a legal string or numeric variable name and cannot be previously dimensioned. However, it may be previously defined. The latest definition takes precedence. The argument list "(arg1 [arg2, . . . . ,arg3] )" must be supplied to indicate a function. NOTE: The arguments are real, not dummy variables, and do change as evaluation proceeds.

```
10 REM DEFINE A SQUARE FUNCTION
20 DEF FN S1(I) = I * I
30 PRINT FN S1(3),I,FN S1(5),I
40 END
```

```
*RUN
```

```
9 3 25 5
```

```
END AT LINE 40
```

```
*
```

### END

The END statement causes control to return to the command mode. An END statement message is typed, giving the line number of the END statement. END also causes the next statement pointer to be set to the beginning of the program so a CONTINUE resumes execution at the beginning of the program.



An END statement may appear anywhere in the program, as many times as desired. If a program does not contain an END statement, it "runs off the end." In this case, BASIC generates a pseudo end statement at line 65,535.

## INPUT AND LINE INPUT

The INPUT statement is used when data is to be READ from the terminal keyboard or from a mass storage device **working through the console terminal**. The form of the INPUT statement is:

```
INPUT prompt;var1, . . . , varn
```

If the first element in the list following the INPUT statement is a string, INPUT assumes it is a PROMPT and types the string in place of a question mark (?). *If no prompt string is desired but the first variable is a string variable, a leading semicolon is inserted. For example:*

```
INPUT ;S3$(2)
```

*This statement tells BASIC that the data to come from the console terminal is to be placed in a dimensioned string named S3.*

Data input from the console terminal has a format identical to the DATA statement.

**NOTE:** Responses to string inputs must be enclosed in quotes.

Expressions may be supplied and null fields cause the variable to retain its previous value. If the user response does not supply sufficient data to complete the INPUT statement, another "?" prompt is issued, requesting more data input at the terminal. **CAUTION:** If you supply too much data, it will be ignored. The next INPUT statement issues a fresh READ to the terminal.

*The response to the LINE INPUT statement cannot be continued on another line, as they are terminated by the return key.*

When there are several values to be entered via the input statement, it is helpful to print a message explaining the data needed, using the prompt string. For example:

```
10 INPUT "THE TIME IS";T
```

When this line of the program is executed, BASIC prints

```
THE TIME IS
```

and then waits for a response.

The LINE INPUT statement is used to input one line of string data from the console terminal and assign it to a string variable. Its form is identical to the INPUT form, but the string should not be enclosed in quotes.

## STOP

The STOP statement causes BASIC to enter the command mode. The message stating the line number of the STOP is printed. The next line pointer is left after the STOP statement, so a CONTINUE statement causes execution to resume on the line immediately after the STOP statement. The STOP statement is of the form:

```
STOP
```

The STOP statement can occur several times throughout a single program with conditional jumps determining the actual end of the program. The following example uses the STOP statement to examine a variable during execution.

```
10 A=1:B=2:C=3
20 PRINT A,B,C
30 END
```

```
*RUN
```

```
1 2 3
```

```
END AT LINE 30
```

```
*15STOP
```

```
*RUN
```

```
STOP AT LINE 15
```

```
*PRINT A
```

```
1
```

```
*15
```

```
Stop deleted
```

```
*RUN
```

```
1 2 3
```

```
END AT LINE 30
```

```
*
```

## PREDEFINED FUNCTIONS

### Introduction

There are 26 predefined functions in EX. B.H. BASIC and 16 predefined functions in B.H. BASIC. These functions perform standard mathematical operations such as square roots, logarithms, string manipulation, and special features. Each function has an abbreviated three- or four-letter name, followed by an argument in parentheses. As these functions are predefined, they may be used throughout a program when required. Predefined functions use numeric expressions (nexp), integer expressions (iexp), and in EX. B.H. BASIC, string expressions (sexp). Function key words are automatically followed by an open parenthesis "(".

The abbreviation (narg) is used to indicate a numeric argument, a decimal number lying in the approximate range of  $10^{-38}$  to  $10^{+37}$ . Certain functions do not permit the argument to assume this wide range, as indicated in the function description.

The predefined functions may be used in either the command or program mode.

### Arithmetic and Special Feature Functions

The Arithmetic and Special Feature Functions supported by BASIC are identical for EX. B.H. BASIC and B.H. BASIC with the exception of the functions Maximum, Minimum, Space, Tab, and Tangent. These are discussed later in this section. (See Page 17-67.)

#### THE ABSOLUTE VALUE FUNCTION, ABS (nexp)

The ABSOLUTE VALUE Function gives the absolute value of the argument. The absolute value is the positive portion of the numeric expression. For example:

```
*PRINT ABS(-5.5)
```

```
5.5
```

or,

```
*PRINT ABS(SIN(3.5))
```

```
.350783
```

```
*
```

NOTE: The sine of 3.5 radians is  $-.350783$ .



**THE ARC TANGENT FUNCTION, ATN (nexp)**

The ARC TANGENT Function returns the arc tangent of the argument. For example:

```
*PRINT ATN(1/1)*57.296;"DEGREES"
45.0001 DEGREES
*PRINT 4*ATN(1)
3.14159
*
```

NOTE:  $\pi = 3.14159$

**THE COSINE FUNCTION, COS (nexp)**

The COSINE function returns the COSINE of the argument (nexp) expressed in radians. For example:

```
*PRINT COS(60/57.296)
.500003
*
```

**THE EXPONENTIAL FUNCTION EXP (NEXP)**

The EXPONENTIAL function returns the value  $e^{nexp}$ . If "nexp" exceeds 88, an overflow error is flagged, as the result exceeds  $10^{38}$ . If "nexp" is less than -88, an overflow error occurs. An example of the exponential function is:

```
*PRINT EXP(1),EXP(2),EXP(COS(60/57.296))
2.71828 7.38905 1.64873
*
```

**THE INTEGER FUNCTION, INT (narg)**

The INTEGER function returns the value of the greatest integer value, not greater than "narg." If the argument is a negative number, the INTEGER function returns the negative number with the same or smaller absolute value. For example:

```
*PRINT INT (38.55)
38
*PRINT INT (-3.3)
-3
```

**THE LOGARITHM FUNCTION, LOG (nexp)**

The LOGARITHM function returns the natural logarithm (LOG to the base e) of the argument. You can find the Logarithms of a number N in any other base by using the formula:

$$\text{LOG}_a N = \text{LOG}_e N / \text{LOG}_e a$$

where "a" represents the desired base. Most frequently, "a" is 10 when you are converting to common logarithms. For example:

```
PRINT "A POWER RATIO OF 2 IS";10(LOG(2)/LOG(10));"DECIBELS"
 A POWER RATIO OF 2 IS 3.0103 DECIBELS
*
```

### THE PAD FUNCTION, PAD (0)

The PAD function returns the value of the keypad pressed on the H8 front panel. For example:

```
*PRINT PAD(0)
 6 The #6 key was pressed.
```

The PAD function uses all the front panel debounce and repeat software contained in PAM-8. (See "The Segment Function," Page 17-65, for an additional example.)

NOTE: The PAD function must be completely executed before any other function will respond. Therefore, CONTROL-C, etc., will not work until you press an H8 front panel key.

### THE PEEK FUNCTION, PEEK (iexp)

The PEEK function returns the numeric value of the byte at memory location iexp.

### THE PIN FUNCTION, PIN (iexp)

The PIN function returns the value input from port "iexp" where iexp is a decimal expression ranging from 0-255. For example:

```
*A=PIN(38)
```

Where "A" now contains the data that was at port #38 (46 octal).

### THE POSITION FUNCTION, POS (0)

The POSITION function returns the current terminal printhead (cursor) position. Although the numeric argument (0) is ignored, it must be present to complete the function. The value returned is a decimal number indicating the column number of the printhead (cursor) position. For example:

```
*PRINT POS(0), POS(0), POS(0); POS(0); POS(0)
 1 15 29 33 37
*
```

### THE RANDOM FUNCTION, RND (narg)

The RANDOM number function returns the next element in a pseudo random series. The RANDOM number generator is not truly random, and may be manipulated by controlling the argument. If narg>0, the random number generator

returns the next random number in the series. If  $narg = 0$ , the random number generator returns the previously returned random number. If  $narg < 0$ , the value "narg" is used as a new seed for a random number, thus starting an entire new series. Using these three inputs to the random number series, the programmer may continuously return the same number while de-bugging the program, determine what the series of numbers will be when the program is run, or start a series of new random numbers each time BASIC is loaded. For example:

```
10 FOR A=0 TO 2
20 PRINT RND(1)
30 NEXT
40 END
```

```
*RUN
.93677
.566681
.53128
```

```
END AT LINE 40
```

```
*RUN
.564484
.787262
.332306
```

```
END AT LINE 40
```

```
*20PRINT RND(0)
```

```
*RUN
.332306
.332306
.332306
```

```
END AT LINE 40
```

```
*20PRINT RND(-1)
```

```
*RUN
6.25305E-02
6.25305E-02
6.25305E-02
```

```
END AT LINE 40
```

```
*20PRINT RND(-5)
```

```
*RUN
.460968
.460968
.460968
```

```
END AT LINE 40
```

```
*
```



## THE SEGMENT FUNCTION, SEG (narg)

The SEG function returns a numeric value which is the correct 8-bit binary number to display the digit on the H8 front panel LED's. The argument must be an integer between 0 and 9. The following program demonstrates the use of PAD, POKE, and SEG in EX. B.H. BASIC. See the second example for a B.H. BASIC program.

```
10 REM A PROGRAM TO USE THE FRONT PANEL LEDS. CNTRL 2,1 TURNS
20 REM ON THE LEDS WITHOUT UPDATE. THE KEYPAD NOW DRIVES THE
30 REM DISPLAY THRU BASIC. 8203 IS THE FIRST LED MEM LOCATION.
40 CNTRL 2,1
50 A=8203
60 FOR I=A TO A+8
70 POKE I,SEG(PAD(0))
80 NEXT I
90 GOTO 60
*RUN
```

When this program is executed, the H8 front panel LEDs respond to the H8 keypad numeric entries. To escape from the program you would type CONTROL-C and then press a key on the H8 front panel. NOTE: If BASIC is to be used, the EX. B.H. BASIC command CNTRL cannot be used to turn on the displays and turn off the update. Therefore, the program is modified to use a POKE command at the first line which stops display update. The program escape routine is still the same. When you are using BASIC, the program is:

```
40 POKE 8200,2
50 A=8203
60 FOR I=A TO A+8
70 POKE I,SEG (PAD(0))
80 NEXT I
90 GOTO 60
```

\*RUN

\*

**THE SIGN FUNCTION, SGN (narg)**

The SIGN function returns the value +1 if "narg" is a positive value, 0 if "narg" is 0, and -1 if "narg" is negative. For example:

```
*PRINT SGN(5.6)
```

```
1
```

```
*PRINT SGN(-500)
```

```
-1
```

```
*PRINT SGN(12-12)
```

```
0
```

```
*
```

**THE SINE FUNCTION SIN (nexp)**

The SIN function returns the sine of the argument (nexp) expressed in radians. For example:

```
*PRINT SIN(30/57.296)
```

```
.499999
```

```
*
```

**SQUARE ROOT FUNCTION, SQR (narg)**

The SQUARE ROOT function returns the square root of "narg." The argument "narg" must be greater than or equal to 0 (for example, positive).

```
*FOR A=0 TO 5:PRINT A,SQR(A),A*A:NEXT
```

|   |         |    |
|---|---------|----|
| 0 | 0       | 0  |
| 1 | 1       | 1  |
| 2 | 1.41421 | 4  |
| 3 | 1.73205 | 9  |
| 4 | 2       | 16 |
| 5 | 2.23607 | 25 |

```
*
```

**USER DEFINED FUNCTION, USR (narg)**

This function calls a user-supplied machine language function. The user-supplied function should be stored in high memory above BASICs user set high memory limit. You should place the starting address of your machine language function at location USRFCN. (See "Appendix C.") A RET instruction exits from the user-defined function. An example of the USR function is given in "Appendix E," Page 17-111.

**THE FREE SPACE FUNCTION, FRE (0) [B.H. BASIC ONLY]**

The FREE function returns the number of free bytes of program and variable storage area available. You must supply the dummy argument 0 to complete the function. This function is not available in EX. B.H. BASIC. Most frequently, this function is used in the command mode in conjunction with a PRINT statement. For example:

```
*PRINT FRE (0)
1224
*
```

**THE MAXIMUM FUNCTION, MAX (nexp1, . . . ,nexpn)**

The MAXIMUM function returns the maximum value of all the expressions which are arguments of the function. For example:

```
10 LET A=1
20 PRINT MAX(COS(A),SIN(A)/COS(A))
30 END
*RUN
1.55741
END AT LINE 30
*
```

The expression containing the maximum value is the expression for the tangent of 1 radian, (1.55741).

**THE MINIMUM FUNCTION, MIN (nexp1, . . . ,nexpn)**

The MINIMUM function returns the lowest value of all the expressions contained in the argument. For example:

```
*PRINT MIN(1,2,3,4,.5)
.5
*
```

**THE TANGENT FUNCTION, TAN (nexp)**

The TANGENT Function returns the TANGENT of the argument "nexp" expressed in radians. For example:

```
*PRINT TAN (45/57.296)
.999996
*
```



**THE SPACE FUNCTION, SPC (iexp)**

The SPACE function spaces the printhead (cursor) iexp spaces to the right of its present position. For example:

```
*PRINT 12,14,SPC(20);600
 12 14 600
*
```

**THE TAB FUNCTION TAB (iexp)**

The TAB function moves the printhead (cursor) to the iexp th column. NOTE: If the printhead is at or past the iexp th column, the function is ignored. For example:

```
*PRINT TAB(20);60,70
* 60 70
```

**String Functions (Extended BASIC Only)**

*Extended BASIC contains various functions for processing character strings in addition to the functions used for mathematical operations. These functions allow the program to concatenate two strings, access a part of string, generate a character string corresponding to a given number, or generate a number for a given string.*

**THE CHARACTER FUNCTION, CHR\$ (iexp)**

*The CHARACTER function returns a string that consists of a single character. The character generated has the ASCII code "iexp." NOTE: "iexp" is a decimal number and must be converted to octal for comparison with most ASCII character tables. See "Appendix D" on Page 0-52 of the "Introduction." For example:*

```
*PRINT CHR$(65)
A
*PRINT CHR$(70)
F
*
```

**NOTE:** If iexp = 0, the generated string is null.

## THE STRING FUNCTIONS, STR\$ (narg)

The *STRING* function encodes the argument (*narg*) into ASCII in the same format used by the *PRINT* statement for numbers. These characters are returned as a string, with leading and trailing blanks. For example:

```
*PRINT STR$(100)
100
*PRINT "100"
100
*
```

|   |                        |
|---|------------------------|
| } | STR\$ function         |
| } | Normal string printing |

## THE ASCII FUNCTION, ASC (sexp)

The *ASCII* function returns the ASCII code for the first character in the string expression (*sexp*). If the string is a null, the *ASCII* function returns a zero. The return is a decimal number and must be converted to octal for comparison to most ASCII tables. See "Appendix D" on Page 0-52 of the "Introduction." For example:

```
*PRINT ASC("ABC")
65
*PRINT CHR$(65)
A
*
```

## THE LEFT STRING FUNCTION, LEFT\$ (sexp, iexp)

The *LEFT STRING* function returns the "*iexp*" left-most characters of the string expression (*sexp*). If "*iexp*" equals 0, the null string is returned. For example:

```
*PRINT LEFT$("HELLO, THIS IS A TEST",10)
HELLO, THI
*
```

## THE RIGHT STRING FUNCTION, RIGHT\$ (sexp, iexp)

The *RIGHT STRING* function returns the right-most "*iexp*" characters of the string expression (*sexp*). If "*iexp*" equals 0, the null string is returned. For example:

```
*PRINT RIGHT$("HELLO, THIS IS A TEST",10)
IS A TEST
*
```

**THE MIDDLE STRING FUNCTION, MID\$ (sexp, iexp [, iexp2] )**

The **MIDDLE STRING** function returns the right-hand substring of the string expression "sexp" starting with the "iexp1" th character from the left-hand side (the first character is 1). The return continues for "iexp2" characters or to the end of the string if the optional terminating expression "iexp2" is omitted. For example:

```
*PRINT MID$("HELLO, THIS IS A TEST",10,10)
IS IS A TE
*
```

**THE NUMERIC VALUE FUNCTION, VAL (sexp)**

The **NUMERIC VALUE** function returns the numeric value of the number encoded in the string expression (sexp). For example:

```
*PRINT VAL (".003E-1")
3.00000E-04
*
```

**THE LEN FUNCTION, LEN (sexp)**

The **LEN** function returns the length of the string expression "SEXP." For example:

```
*PRINT LEN("HOW LONG IS THE STRING?")
23
```



## EDITING COMMANDS

BENTON HARBOR BASIC provides several commands used to halt program execution, erase characters, delete lines, add lines, and provide other editing functions. A great number of these editing commands are common to all the Heath H8 Software packages. Their operation is covered in detail in the "Console Driver" section (Page 0-36) of the "Introduction" to this Software Reference Manual.

### Control-C, CNTRL-C

CONTROL-C is a general-purpose cancel key. It can be used to stop a mass storage input or output operation, stop program execution, stop a listing, and to stop a program during an input statement. Using CONTROL-C results in the CC error message (CNTRL-C in EXTENDED BASIC). NOTE: A CONTROL-C causes the program to terminate at the end of a current statement.

You can continue program execution by using the CONTINUE statement.

### Inputting Control

The following control characters take effect when you are inputting information from the control terminal.

#### BACKSPACE, BKSP/CNTRL-H

The BACKSPACE key (or a CONTROL-H) causes a one-character backspace. The backspace code is echoed to the terminal so devices with backspace capability physically backspace. Attempting to backspace into column zero is illegal and causes a terminal bell code to be echoed. NOTE: Backspace can be changed at configuration. See "Product Installation" on Page 0-19 of the "Introduction" to this Software Reference Manual.

#### RUBOUT

The RUBOUT key causes BASIC to discard the current line being inputted. A carriage-return feed line is sent to the console terminal, and the user may now re-enter the entire line. NOTE: Rubout can be changed at configuration. See "Product Installation" on Page 0-19 of the "Introduction" to this Software Reference Manual.

## Outputting Control

The following control characters take effect only when you are outputting information to the console terminal. They should not be used when you are inputting information via the console terminal, as they affect characters being echoed to the console.

### OUTPUT SUSPENSION AND RESTORATION, CNTRL-S AND CNTRL-Q

Type CONTROL-S to suspend the output and to suspend program execution. This command is particularly useful when you are using a video terminal; you can use the CONTROL-S (suspend) feature each time a screen is nearly filled and information at the top will be lost due to scrolling.

By typing CONTROL-Q, you permit BASIC to continue execution and outputting information to the terminal. CONTROL-Q cancels the CONTROL-S function.

### The DISCARD FLAG, CNTRL-O and CNTRL-P

Type a CONTROL-O to toggle the DISCARD FLAG. Setting the DISCARD FLAG stops output on the terminal but does not halt program execution until you retype CONTROL-O or until you type a CONTROL-P to clear the DISCARD FLAG. CONTROL-O is often used to discard the remainder of long listings and other similar outputs. BASIC clears the discard flag when it returns to the command mode or when an INPUT statement is executed, so that the prompt will appear.

## Command Completion

When you are inputting information from the console terminal in the command mode, or in response to an INPUT command, BASIC checks the incoming characters for the initial characters of a keyword. As soon as enough characters of a keyword are entered to uniquely identify it, and it is distinguished from a variable name, BASIC completes the keyword into the terminal. For example, to enter the command SCRATCH, type SC. Since SC uniquely determines SCRATCH and SC is not a legal variable name, BASIC types the characters RATCH immediately following the SC. Striking the backspace key backspaces over the entire word SCRATCH.

Function keywords are automatically followed by an open parenthesis "(" . Other keywords are immediately followed by a blank.



## Enforced Lexical Rules

BASIC enforces two lexical rules during input.

1. Two adjacent alphabetical characters must start a keyword. For example, XX is illegal as no keyword starts with "XX" and "XX" is an illegal variable name. This rule is excepted when following a REMARK statement or when the characters are contained within quotes (indicating a string).
2. A quoted string must be closed; every quote character must have a mate on the same line.

Should a character be typed which is in violation of these rules, a bell code is echoed and the character is ignored.

## General Text Rules

### BLANKS

BASIC programs are generally "free format." That is, blanks (spaces) may be included freely with the following restrictions.

1. Variable names, keywords, and numeric constants may not contain imbedded blanks.
2. Blanks may not appear before a statement number.

### LINE INSERTION

You can insert lines into a BASIC program by simply typing an appropriate line number followed by the desired line of text. This is done in response to the command mode prompt (an asterisk). Except when running a program, BASIC remains in the command mode, showing a single asterisk (\*) as a prompt. NOTE: The text should immediately follow the last digit of the line number. Although intervening blanks are allowable, they waste memory. BASIC automatically inserts a blank when listing the text. For example:

```
*100PRINT "HEATH BASIC"
LIST
100 PRINT "HEATH BASIC"
```



## LINE LENGTH

A line in BENTON HARBOR BASIC is restricted to 80 characters, and lines in Extended BENTON HARBOR BASIC are restricted to 100 characters. This restriction on line length is completely independent of the console width, which was established by the software configuration procedure (see Page 0-19 of the "Introduction" to this Software Reference Manual). NOTE: If the console terminal, for example, was set at a width of 34 characters, the console will display three complete lines for one line of BASIC.

## LINE REPLACEMENT

Replace existing program lines by simply typing the line number and the new text. This is the same process you use to insert a new line. The old line is completely lost once the new line is entered.

## LINE DELETION

Delete lines by typing the line number immediately followed by a carriage-return. You can leave blank lines by typing the single space before typing the carriage-return.

## ERRORS

BASIC detects many different error conditions. When an error is detected, a message of the form:

```
! ERROR-(ERROR MESSAGE) [at line NNNNN]
```

is typed. BASIC returns to the command mode (if it is not already in the command mode), ringing the console terminal bell. If BASIC is in the command mode, the "at line NNNN" portion of the error message is omitted. For example:

```
*PRINT 1/0
! ERROR - /0
*10PRINT 1/0
*RUN

! ERROR - /0 AT LINE 10
*
```

NOTE: If a line of BASIC contains an error, you can correct it by retyping the entire line. Once the line number is typed, the contents of the old line are lost. To delete a line, type the line number and follow it with a carriage-return.

### Error Messages

The following "Basic Error Table" describes the error messages generated by BENTON HARBOR BASIC and Extended BENTON HARBOR BASIC. Two columns of error messages are given, as BENTON HARBOR BASIC provides a shortened form of the error messages. An explanation is given after each error message in the Comments column.

### Recovering from Errors

When an error is detected, BASIC enters the command mode with the variables and stacks as they were at the time of the error. Thus, the user can use PRINT and LET statements to examine and alter variable contents. Likewise, a GOTO statement can be used to set the next statement pointer to any desired statement number. Often, a combination of these techniques allows the user to continue a program with the error corrected.

NOTE: If program text in an EX. B.H. BASIC program is modified in any way, the GOSUB and FOR stacks are purged. If an error occurred in a GOSUB routine, or a FOR LOOP, the entire program must be restarted. If you modify text in a B.H. BASIC program, B.H. BASIC CLEARs all variables.

## ERRORS

There are many different error conditions. When an error is detected, a message is printed in the form:

ERROR (ERROR MESSAGE) (AT LINE NUMBER)

The BASIC interpreter to the command mode (if it is not already in the command mode), printing the console format. If BASIC is in the command mode, the "LINE NUMBER" portion of the error message is omitted. For example:

ERROR 123  
LINE 100  
END

ERROR - NO AT LINE 100

If the BASIC interpreter contains an error, you can correct it by retyping the line. Once the line number is typed, the contents of the old line is lost. To correct the line type the line number and follow it with a carriage-return.

## Error Messages

The following "Error Table" describes the error messages generated by the BASIC interpreter. The messages are given in two columns. The first column of error messages are given as BENTON HARBOR BASIC provides a detailed form of the error messages. An explanation is given after each error message in the Comments column.

## Recovering from Errors

When an error is detected, BASIC enters the command mode with the variable pointer set to the line of the error. Thus, the user can use PRINT and GOTO statements to examine and alter variable contents. Likewise, a GOTO statement can be used to set the next statement pointer to any desired statement. Often, a combination of these techniques allows the user to continue a program with the error corrected.

If a program is in an EX-BASIC program is modified in any way, the GOSUB routine is required. If an error occurred in a GOSUB routine, or a FOR-LOOP, the program must be restarted. If you modify text in a B-H BASIC program, B-H BASIC will erase all variables.



## BASIC ERROR TABLE

| BASIC | EXTENDED<br>BASIC | COMMENTS                                                                                                                                                          |
|-------|-------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| AC    | ARG CT            | Argument count. Incorrect number of arguments supplied to a DEF defined function.                                                                                 |
| CC    | CNTRL-C           | CONTROL-C. Program execution or other operation aborted by a CONTROL-C typed at the console terminal.                                                             |
| DE    | NO DAT            | Data exhausted. A READ called for more data than is available in the DATA statement.                                                                              |
| /0    | /0                | Divide by zero. An attempt to divide by zero. NOTE: Either dividing by the number zero or dividing by an expression which evaluates to zero generates this error. |
| IN    | NUM               | Illegal number. The line number referenced by a command or program statement is not used by BASIC.                                                                |
| IU    | USE               | Illegal usage. A command or statement is used in the improper context.                                                                                            |
| NX    | NXT               | Next variable missing. No FOR statement matching the accompanying NEXT statement.                                                                                 |
| OV    | OVRFL             | Overflow. Memory space is filled by program text.                                                                                                                 |
| RE    | RTN               | Return error. A RETURN is encountered without a calling statement.                                                                                                |
| S#    | STAT#             | Statement number. The referenced statement number does not exist in the program.                                                                                  |

| BASIC | EXTENDED<br>BASIC | COMMENTS                                                                                                                                                                                           |
|-------|-------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SY    | SYN               | Syntax error. Command, statement, or function uses incorrect separators, functions, etc..                                                                                                          |
| MO    | MEM OV            | Table overflow. One of the internal tables has grown too large for memory. Check GOSUBs, FOR loops, and DIM.                                                                                       |
| SR    | SUBS RANG         | Subscript range. The subscript size of a dimensioned variable exceeded the size defined by the DIM statement.                                                                                      |
| SC    | SUBS CNT          | Subscript count. The number of subscripts assigned to a variable exceeds the number defined in the DIM statement.                                                                                  |
| ND    | DIM?              | Not dimensioned. The subscripted variable has not been dimensioned in a DIM statement.                                                                                                             |
| IC    | ILL CHA           | Illegal character. An improper character is assigned to a command or function<br>NOTE: In B.H. BASIC, an attempt to assign a string to a numeric variable results in an illegal character message. |
| FU    | FN ?              | Function error. No single line function defined by a DEF statement was found when the FN function was encountered. NOTE: The DEF FN must be executed prior to executing the FN function.           |
| TE    | TAPE              | Tape error. An error in handling the mass storage device at the load dump port.                                                                                                                    |

| BASIC | EXTENDED<br>BASIC | COMMENTS                                                                                                           |
|-------|-------------------|--------------------------------------------------------------------------------------------------------------------|
|       | CNTRL-B           | <i>CONTROL-B error. A CONTROL-B entered from the console terminal, but no CONTROL-B processing in the program.</i> |
|       | STR LE            | <i>String length error. The length of a string exceeded 255 characters.</i>                                        |
|       | TYP CNFLC         | <i>Type conflict. String data supplied for a numeric variable or numeric data supplied for a string variable.</i>  |





## APPENDIX A

### Loading Procedures

#### Loading From the Software Distribution Tape

1. Load the tape in the reader.
2. Ready the tape transport.
3. Press LOAD on the H8 front panel.
4. A single beep indicates a successful load.
5. Press GO on the H8 front panel.
6. The console terminal will respond with:

```
HEATH/WINTEK H8 BASIC
BENTON HARBOR BASIC ISSUE # 05.01.00
COPYRIGHT 01/77 BY WINTEK CORP.
```

7. Configure B.H. BASIC or EX. B.H. BASIC as desired, answering the following questions. Prompt each question by typing its first character on the console terminal keyboard.

```
•AUTO NEW-LINE (Y/N)?
•BKSP = 00008/
•CONSOLE LENGTH = 00080/
•HIGH MEMORY = 16383/
•LOWER CASE (Y/N)?
•PAD = 4/
•RUBOUT = 00127/
•SAVE?
•
```

8. Before executing SAVE, have the tape transport ready at the DUMP port.
9. To use BASIC directly from the distribution tape, type the return key at any time rather than a question prompt key. The Console Terminal will display:

```
B.H. BASIC # 05.01.00
```

```
*
```

BASIC is ready to use.

## Loading From a Configured Tape

1. Load the tape in the tape transport.
2. Ready the tape transport.
3. Press LOAD on the H8 front panel.
4. A single beep indicates a successful load.
5. Press GO on the H8 front panel.
6. The console terminal responds with:

```
B.H. BASIC # 05.00.00
```

```
*
```

BASIC is ready to use in the configured form.



## APPENDIX B

### A Summary of BASIC

For additional details, refer to the page number that is given with each of the following topics.

#### See Page

#### Numeric Data

17-9

Numbers may be real or integer with the following characteristics:

Range .....  $10^{-38}$  to  $10^{+37}$ .  
Accuracy ..... 6.9 digits.  
Decimal range ..... 0.1 to 999999.  
Exponential format .....  $(\pm) X.XXXXXXE (\pm) NN$ .

#### Boolean Data

17-10

Integer numbers from 0 to 65535 represent two byte binary data from 00000000 00000000 to 11111111 11111111. Fractional parts of numbers between 0 and 65535 are discarded.

#### String Data (Extended BASIC Only)

17-10

Data is all printed in ASCII characters plus the BELL, BLANK and LINE FEED, with the following characteristics:

Maximum string length ..... 255 characters.  
Enclosure ..... Quotation marks (") on both ends.  
Multiple lines ..... Not allowed for a single string.

#### Variables

17-11

Variables are named by a single letter (A through Z), or a single letter followed by a single number (0 through 9). For example: A or A6.

See Page

## Subscripted Variables

17-12

Subscripted variables are named like variables, but are followed by dimensions in parentheses. Subscripted variables are of the form:

$A(N_1, N_2, \dots, N_x)$  For example:  $A(1, 2, 7)$  or  $A6(1, 5)$ .

You must use a DIMENSION statement to define the range and number of allowable subscripts for a variable.

## Arithmetic Operators

17-14

Listed in order of priority. Operators on the same line have equal precedence. Parenthetical operations are performed first. Precedence is left to right if all other factors are equal.

| <u>SYMBOL</u> | <u>EXPLANATION</u>                |
|---------------|-----------------------------------|
| -             | Unary negation logical compliment |
| ↑             | Exponentiation. Ex BASIC only     |
| * /           | Multiplication division           |
| + -           | Addition subtraction              |

## Relational Operators

17-18

| <u>SYMBOL</u> | <u>EXPLANATION</u>       |
|---------------|--------------------------|
| =             | Equal to                 |
| <             | Less than                |
| <=            | Less than or equal to    |
| >             | Greater than             |
| >=            | Greater than or equal to |
| <>            | Not equal to             |

See Page

## Boolean Operators

17-19

Boolean operators perform the Boolean (logical) operations on two integer operands. The operands must evaluate to integers in the range of 0 to 65535. The operators are:

|     |                                |
|-----|--------------------------------|
| NOT | Logical complement, bit by bit |
| OR  | Logical OR, bit by bit         |
| AND | Logical AND, bit by bit        |

## String Variables

17-21

String variables may be either subscripted or nonsubscripted. They take the same form as numeric or Boolean variables but are followed by a dollar sign (\$) to indicate a string variable. For example: A\$ A6\$ A\$(1,2,7) or A6\$(1,5).

## String Operators

17-22

String expressions may be operated on by the relational operators as well as the plus (+) symbol. The plus symbol is used to perform string concatenation.

## Line Numbers

17-25

When it is used in the program mode, BASIC requires that each line be preceded by an integer line number in the range 1 to 65535.

## The Command Mode

17-23

The command mode does not use line numbers. Statements are executed when a carriage-return is typed.

## Multiple Statements on One Line

17-25★

BASIC permits multiple statements on one line. Each statement is separated from the others by a colon (:). DATA statements may not appear on lines with other statements.

★See "Basic Statements."



## Command Mode Statements

| <u>COMMAND</u> | <u>FORM</u>                    | <u>DESCRIPTION</u>                                                                                                                 | <u>SEE Pg.</u> |
|----------------|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------|----------------|
| BUILD          | BUILD <i>iexp1, iexp2</i>      | Automatically generates program line numbers starting at <i>iexp1</i> in steps of <i>iexp2</i> .                                   | 17-27          |
| CONTINUE       | CONTINUE                       | Resumes program execution.                                                                                                         | 17-27          |
| DELETE         | DELETE [ <i>iexp1, iexp2</i> ] | Deletes program lines between <i>iexp1</i> and <i>iexp2</i> .                                                                      | 17-28          |
| DUMP           | DUMP "name"                    | Saves current program "name" on mass storage media at load dump port; "name" is up to 80 ASCII characters.                         | 17-28          |
| LOAD           | LOAD "name"                    | Loads program "name" from mass storage media at load dump port; "name" is up to 80 ASCII characters. Current program is destroyed. | 17-29          |
| RUN            | RUN                            | Start execution of current program. Preclears all variables, stacks, etc..                                                         | 17-30          |
| SCRATCH        | SCRATCH<br>SURE?Y              | Clears all program and data storage area. Any response to SURE but Y cancels SCRATCH.                                              | 17-31          |
| VERIFY         | VERIFY "name"                  | Performs a checksum on the mass storage record titled "name." No response if record is bad.                                        | 17-31          |

## Command and Program Mode Statements

| <u>COMMAND</u> | <u>FORM</u>                                                    | <u>DESCRIPTION</u>                                                                                                                                                                                                                                                                            | <u>SEE Pg.</u> |
|----------------|----------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| CLEAR          | CLEAR [varname]                                                | Clears all variables, arrays, string buffers, etc.<br><i>Optionally clears named variable (varname).</i><br><i>Specifies functions and arrays as V(.</i>                                                                                                                                      | 17-33          |
| CONTROL        | CNTRL iexp1, iexp2                                             | CNTRL 0 sets a GOSUB to line iexp2<br>when a CONTROL-B is typed.                                                                                                                                                                                                                              | 17-34          |
|                |                                                                | CNTRL 1 sets iexp2 digits<br>before exponential format is used.                                                                                                                                                                                                                               | 17-35          |
|                |                                                                | CNTRL 2 controls the H8 front<br>panel. If iexp2: = 0, display off;<br>=1, display on without update;<br>=2, display on with update.                                                                                                                                                          | 17-35          |
|                |                                                                | CNTRL 3 sets the width of a<br>print zone to iexp2 columns.                                                                                                                                                                                                                                   | 17-35          |
|                |                                                                | CNTRL 4 controls the H8 hardware<br>clock. iexp2 = 0, clock off. iexp2 = 1,<br>clock on.                                                                                                                                                                                                      | 17-36          |
| DIMENSION      | DIMvarname(iexp1 [, . . . . ,iexpn ] ) [,varname2( . . . . ) ] | Defines the maximum size of<br>variable arrays.                                                                                                                                                                                                                                               | 17-36          |
| FOR/NEXT       | FOR var = nexp1 TO nexp2 [STEP nexp3]                          |                                                                                                                                                                                                                                                                                               |                |
|                | NEXT var                                                       | Defines a program loop. Var is<br>initially set to nexp1. Loop<br>cycles until NEXT is executed;<br>then var is incremented by<br>nexp3 (default is +1). Looping<br>continues until var > nexp2<br>(or less than nexp2 if STEP is<br>negative). The statement after<br>NEXT is then executed. | 17-37          |

| <u>COMMAND</u>   | <u>FORM</u>                                                | <u>DESCRIPTION</u>                                                                                                                                                    | <u>SEE Pg.</u> |
|------------------|------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| FREE             | FREE                                                       | Displays the amount of memory assigned to tables and text                                                                                                             | 17-41          |
| GOSUB/<br>RETURN | GOSUB iexp<br>RETURN                                       | Transfers execution sequence of program to line iexp (the beginning of a subroutine). RETURN returns execution sequence to the statement following the calling GOSUB. | 17-42          |
| GOTO             | GOTO iexp                                                  | Unconditionally transfers the program execution sequence to the line iexp.                                                                                            | 17-44          |
| IF/THEN          | IF expression THEN<br>iexp IF expression<br>THEN statement | If the expression is true, control passes to iexp line or to "statement." If the relation is false, control passes to the next independent statement.                 | 17-45          |
| LET              | LET var = nexp<br>LET var\$ = sexp                         | Assigns the value nexp (or sexp in the case of strings) to the variable var (or var\$). LET keyword is optional.                                                      | 17-46          |
| LIST             | LIST[iexp1] [,iexp2]                                       | Lists the entire program on the console terminal. Lists the line iexp1 or the range of lines iexp1 to iexp2.                                                          | 17-47          |
| ON/GOSUB         | ON iexp1 GOSUB<br>iexp2, ..., iexpn.                       | Permits a computed GOSUB. iexp1 is evaluated and acts as an index to line numbers iexp2 thru iexpn, each pointing to a different subroutine.                          | 17-48          |



| <u>COMMAND</u> | <u>FORM</u>                                     | <u>DESCRIPTION</u>                                                                                                                                                                                                                                                                                                                      | <u>SEE Pg.</u> |
|----------------|-------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| ON/GOTO        | ON iexp1 GOTO<br>iexp2, . . . ,iexpn            | Permits a computed GOTO. iexp1<br>is evaluated and acts as an index<br>to line numbers iexp2 thru iexpn.                                                                                                                                                                                                                                | 17-48          |
| OUT            | OUT iexp1, iexp2                                | Outputs a number iexp2<br>to output port iexp1.                                                                                                                                                                                                                                                                                         | 17-48          |
| PAUSE          | PAUSE ( <i>iexp</i> )                           | Ceases program execution<br>until a console terminal key<br>is typed. <i>Ceases program<br/>execution for 2 X iexp mS.</i>                                                                                                                                                                                                              | 17-49          |
| POKE           | POKE iexp1, iexp2                               | Writes a number iexp2 into<br>memory location iexp1.                                                                                                                                                                                                                                                                                    | 17-49          |
| PORT           | PORT <i>iexp</i>                                | <i>Assigns the print function to<br/>port iexp. Assigns the Console<br/>Terminal function to the<br/>device at port iexp if iexp<br/>is negative.</i>                                                                                                                                                                                   | 17-50          |
| PRINT          | PRINT( <i>nexp. sep1 . . . nexpn(sepn)</i> )    | <i>Prints the value of the expression<br/>(s) exp with a leading and<br/>trailing space. Expressions may be<br/>numeric or string. If the separator<br/>is a comma, the next print zone is<br/>used. If the separator is a semicolon,<br/>no print zones are used. No<br/>separator prints each expression<br/>value on a new line.</i> | 17-50          |
| READ & DATA    | READ var1, . . . ,varn<br>DATA exp1 . . . ,expn | The READ statement assigns the<br>values exp1 thru expn in the<br>data to the variables var1 thru varn.                                                                                                                                                                                                                                 | 17-54          |

| <u>COMMAND</u> | <u>FORM</u> | <u>DESCRIPTION</u>                                                                           | <u>SEE Pg.</u> |
|----------------|-------------|----------------------------------------------------------------------------------------------|----------------|
| REMARK         | REM         | Text following the REM is not executed and is used for commentary only.                      | 17-55          |
| RESTORE        | RESTORE     | Causes the program to reset the DATA pointer, thus reusing data at the first DATA statement. | 17-56          |
| STEP           | STEP iexp   | Executes iexp lines of the program. Then returns BASIC to the command mode.                  | 17-57          |

## Program Mode Statements

|            |                                       |                                                                                                   |       |
|------------|---------------------------------------|---------------------------------------------------------------------------------------------------|-------|
| DEF        | DEF FN varname (arg list) = exp       | Defines a single-line program function created by the user.                                       | 17-58 |
| END        | END                                   | Causes control to return to the command mode.                                                     | 17-58 |
| INPUT      | INPUT prompt; var1, . . . , varn      | Reads data from the console terminal. <i>String data must be enclosed in quotes.</i>              | 17-59 |
| LINE INPUT | LINE INPUT prompt; var1, . . . , varn | Reads string data from the terminal. <i>String data for LINE INPUT is not enclosed in quotes.</i> | 17-59 |
| STOP       | STOP                                  | Causes BASIC to enter the command mode when the statement containing STOP is executed.            | 17-60 |

## Predefined Functions

| <u>FUNCTION</u> | <u>DEFINITION</u>                                                                                                                                                    | <u>SEE Pg.</u> |
|-----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| ABS (nexp)      | Returns the absolute value of nexp.                                                                                                                                  | 17-61          |
| ATN (nexp)      | Returns the arctangent of nexp (radians).                                                                                                                            | 17-62          |
| COS (nexp)      | Returns the cosine of nexp (radians).                                                                                                                                | 17-62          |
| EXP (nexp)      | Returns $e^{nexp}$ .                                                                                                                                                 | 17-62          |
| INT (narg)      | Returns the integer value of narg.                                                                                                                                   | 17-62          |
| LOG (nexp)      | Returns the natural logarithm of nexp.                                                                                                                               | 17-62          |
| PAD (0)         | Returns the value of the H8 front panel key pressed. Includes key debounce.                                                                                          | 17-63          |
| PEEK (iexp)     | Returns the numeric value at memory location iexp.                                                                                                                   | 17-63          |
| PIN (iexp)      | Returns the data input from port iexp.                                                                                                                               | 17-63          |
| POS (0)         | Returns the current terminal printhead (cursor) position (by column number).                                                                                         | 17-63          |
| RND (narg)      | Returns a random number. If narg > 0, RND is next in the series. If narg = 0, RND is the previous random number. If narg < 0, RND algorithm uses narg as a new seed. | 17-63          |
| SEG (narg)      | Returns the correct eight-bit number to display narg (0-9) on the H8 LEDs.                                                                                           | 17-65          |
| SGN (narg)      | Returns +1 if narg is positive.<br>Returns -1 if narg is negative.<br>Returns 0 if narg is zero.                                                                     | 17-66          |
| SIN (nexp)      | Returns the sine of nexp (radians).                                                                                                                                  | 17-66          |
| SPC (iexp)      | Positions printhead (cursor) iexp columns to the right.                                                                                                              | 17-68          |
| SQR (narg)      | Returns the square root of narg.                                                                                                                                     | 17-66          |



| <u>FUNCTION</u>              | <u>DEFINITION</u>                                                                                                                                                                                    | <u>SEE Pg.</u> |
|------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------|
| TAB (iexp)                   | Position printhead (cursor) to the iexp th column.                                                                                                                                                   | 17-68          |
| USR (narg)                   | Calls a user-written machine language function to evaluate narg.                                                                                                                                     | 17-66          |
| FRE (0)                      | Returns the amount of free memory in B.H. BASIC.                                                                                                                                                     | 17-67          |
| MAX (nexp1, . . . ,nexpn)    | Returns the maximum value of expressions nexp1 thru nexpn.                                                                                                                                           | 17-67          |
| MIN (nexp1, . . . ,nexpn)    | Returns the minimum value of expressions nexp1 thru nexpn.                                                                                                                                           | 17-67          |
| TAN (nexp)                   | Returns the tangent of nexp (radians).                                                                                                                                                               | 17-67          |
| CHR\$ (iexp)                 | Returns the ASCII character iexp.                                                                                                                                                                    | 17-68          |
| STR\$ (narg)                 | Returns narg encoded into ASCII with leading and trailing blanks as in the print statement.                                                                                                          | 17-69          |
| ASC (sexp)                   | Returns the ASCII code for the first character in the string sexp.                                                                                                                                   | 17-69          |
| LEFT\$ (sexp, iexp)          | Returns the left iexp characters of the string sexp.                                                                                                                                                 | 17-69          |
| RIGHT\$ (sexp, iexp)         | Returns the right iexp characters of the string sexp.                                                                                                                                                | 17-69          |
| MID\$ (sexp, iexp1) [,iexp2] | Returns the substring of the string sexp starting with the iexp1 th character and ending with the iexp2 th character if iexp2 is specified. If not specified, returns iexp1 th character to the end. | 17-70          |
| VAL (sexp)                   | Returns the numeric value of the number encoded in the string.                                                                                                                                       | 17-70          |
| LEN (sexp)                   | Returns the length of sexp.                                                                                                                                                                          | 17-70          |

## Editing Commands

| <u>COMMAND</u> | <u>FUNCTION</u>                                                                                | <u>SEE Pg.</u> |
|----------------|------------------------------------------------------------------------------------------------|----------------|
| CONTROL-C      | General-purpose cancel. Returns BASIC to monitor mode from any operation or program execution. | 17-71          |
| CONTROL-S      | Suspends the output to the console terminal and suspends program execution.                    | 17-72          |
| CONTROL-Q      | Restores the output to the console terminal and restores program execution.                    | 17-72          |
| CONTROL-O      | Toggles the output discard flag.<br>Does not stop program execution.                           | 17-72          |
| CONTROL-P      | Clears the discard flag set by CONTROL-O.                                                      | 17-72          |





## APPENDIX C

### *Basic Utility Routines*

The following pages contain a description of several utility routines included in BASIC (some are only available in Extended BENTON HARBOR BASIC). They can be used with user-written machine language routines called by the USR function. See "Appendix D" for Utility Routine entry points.

## Appendix C

BASIC - WINTERK BASIC INTERPRETER.  
SELECTED SOURCE LISTING.

HEATH XBASM V1.0 02/18/77  
13:12:38 01-APR-77 PAGE 1

000.000 2 XTEXT MTR

74 \*\*\* BASIC - \*WINTERK\* BASIC INTERPRETER.  
75 \*  
76 \* J. G. LETWIN, 09/76, FOR \*WINTERK\* CORPORATION.  
77 \* LAFAYETTE, IN.

79 \*\*\* COPYRIGHT 09/1976, \*WINTERK\* CORPORATION,  
80 \* 902 N. 9TH ST.  
81 \* LAFAYETTE, IN. 47901

83 \*\* LOW-MEMORY CELLS USED BY BASIC  
84  
040.064 85 ORG 7\*3+.UIVED  
86  
040.064 87 DS 2 ACCX TYPE  
040.066 88 ACCX DS 4  
040.072 89 DS 2 ACCY TYPE  
040.074 90 ACCY DS 4  
91  
92

```

95 ** USR - CALL USER ASSEMBLY LANGUAGE FUNCTION.
96 *
97 * THE *USR* FUNCTION IS ACTUALLY A CALL TO A USER-WRITTEN ROUTINE
98 * WHICH MUST HAVE BEEN PREVIOUSLY LOADED INTO MEMORY BY THE USER.
99 *
100 * THE ADDRESS OF THE FUNCTION'S ENTRY POINT MUST BE IN *USRFCN*.
101 *
102 * BASIC MUST HAVE BEEN PREVIOUSLY CONFIGURED SO THAT THE USER
103 * FUNCTION RESIDES IN MEMORY ABOVE THE STACK POINTER (HIGH MEMORY)
104 * OR ELSE BASIC WILL OVERLAY THE FUNCTION WITH DATA.
105 *
106 * THE FUNCTION IS ENTERED WITH A POINTER TO THE SINGLE ARGUMENT (IN
107 * FLOATING POINT), AND MAY RETURN A FLOATING POINT OR STRING ARGUMENT.
108 * IF NO RETURN VALUE IS PLACED IN *ACCX*, THEN THE ORIGINAL ARGUMENT
109 * REMAINS THERE, AND USR(RETURNS ITS ARGUMENT AS ITS VALUE.
110 *
111 * ENTRY (BC) = $ACCX
112 * EXIT (ACCX) CONTAINS VALUE
113 * USES ALL
114
115
116
117 USRFCN DW 0 ADDRESS OF USER FUNCTION ENTRY POINT
118 * IF = 0, USR(IS NOT LEGAL

```



## Appendix C

BASIC - WINTER BASIC INTERPRETER.  
ERROR PROCESSING

HEATH X8ASM V1.0 02/18/77  
13:12:41 01-APR-77 PAGE 3

```

121 ** ERROR PROCESSING.
122 *
123 * THESE ERROR PROCESSORS ARE ENTERED WHEN AN ERROR IS DETECTED.
124 *
125 * CONTROL PASSES DIRECTLY BACK TO COMMAND MODE.
126 *
127 *
128 ERR.CC EQU * CONTROL-C
129 *
130 ERR.CB EQU * CNTRL-B
131 *
132 ERR.DE EQU * DATA EXHAUSTED
133 *
134 ERR.DO EQU * /O
135 *
136 ERR.IN EQU * ILLEGAL NUMBER
137 *
138 ERR.IU EQU * ILLEGAL USAGE
139 *
140 ERR.NV EQU * NEXT VARIABLE MISSING
141 *
142 ERR.OV EQU * OVERFLOW
143 *
144 ERR.RE EQU * RETURN ERROR
145 *
146 ERR.SL EQU * STRING LENGTH
147 *
148 ERR.SN EQU * STATEMENT NUMBER
149 *
150 ERR.SY EQU * SYNTAX ERROR
151 *
152 ERR.TC EQU * TYPE CONFLICT
153 *
154 ERR.TO EQU * TABLE OVERFLOW
155 *
156 ERR.SR EQU * SUBSCRIPT RANGE
157 *
158 ERR.SC EQU * SUBSCRIPT COUNT
159 *
160 ERR.ND EQU * NOT DIMENSIONED
161 *
162 ERR.IC EQU * ILLEGAL CHARACTER
163 *
164 ERR.UF EQU * UNDEFINED FUNCTION
165 *
166 ERR.TP EQU * TAPE ERROR

```

```

169 ** CVX - COPY VALUE INTO 'X' ACCUMULATOR.
170 *
171 * CVX COPIES A 4 BYTE VALUE INTO THE X ACCUMULATOR.
172 *
173 * ENTRY (DE) = ADDRESS OF VALUE
174 * EXIT COPIED
175 * USES A,F
176
177
178 CVX EQU *
```

```

180 ** CXY - COPY (ACCX) INTO (ACCY)
181 *
182 * ENTRY NONE
183 * EXIT NONE
184 * USES A,F,D,E
185
186
187 CXY EQU *
```

```

189 ** CXV - COPY X TO VALUE.
190 *
191 * CXV COPIES THE CONTENTS OF THE 'X' ACCUMULATOR INTO A MEMORY
192 * LOCATION.
193 *
194 * ENTRY (DE) = TARGET ADDRESS
195 * EXIT COPIED
196 * USES A,F
197
198
199 CXV EQU *
```

```

201 ** IFIX - SPLIT NUMBER INTO INTEGER AND FRACTION.
202 *
203 * IFIX FIXES ((DE)) INTO AN INTEGER.
204 *
205 * ENTRY (DE) = ADDRESS OF NUMBER
206 * EXIT (DE) = INTEGRAL PART OF 0<=N<=65535
207 * TO ERR.IN OTHERWISE
208
209
210 IFIX EQU *
```

BASIC - WINTER BASIC INTERPRETER.  
UTILITY SUBROUTINES

HEATH X8ASM V1.0 02/18/77  
13:12:43 01-APR-77 PAGE 5

```

212 ** IFLT - FLOAT NUMBER.
213 *
214 * ENTRY (DE) = VALUE
215 * EXIT (ACCX) = NUMBER VALUE
216 * (DE) = #ACCX-1
217
218
219 IFLT EQU *
```

```

221 ** TDI - TYPE DECIMAL INTEGER.
222 *
223 * TDI TYPES AN INTEGER AS A 5 PLACE NUMBER. LEADING ZEROS ARE
224 * SUPPRESSED.
225 *
226 * ENTRY (DE) = NUMBER
227 * EXIT TYPED
228 * USES A,F,D,E
229
230
231 TDI EQU *
```

```

233 ** XCY - EXCHANGE (ACCX) WITH (ACCY)
234 *
235 * ENTRY NONE
236 * EXIT NONE
237 * USES A,F
238
239
240 XCY EQU *
```

```

242 ** ZRO - ZERO MEMORY.
243 *
244 * ZRO ZEROS A FIELD OF MEMORY.
245 *
246 * ENTRY (HL) = ADDRESS
247 * (DE) = COUNT
248 * EXIT NONE
249 * USES A,F,D,E,H,L
250
251
252 ZRO EQU *
```



```

255 ** H8 FLOATING POINT FORMAT:
256 *
257 * SINGLE-PRECISION FLOATING POINT NUMBERS ARE REPRESENTED
258 * BY A 4-BYTE VALUE. THE NUMBER CONSISTS OF A 3-BYTE
259 * TWO'S COMPLEMENT MANTISSA, AND A ONE-BYTE BIASED BINARY
260 * EXPONENT. THE NUMBER FORMAT IS:
261 *
262 *
263 * N+0 LEAST SIGNIFICANT MANTISSA BYTE
264 * N+1 MID SIGNIFICANT MANTISSA BYTE
265 * N+2 MOST SIGNIFICANT MANTISSA BYTE
266 * N+3 BIASED EXPONENT
267 *
268 *
269 * EXPONENT:
270 * -----
271 *
272 * EACH FLOATING POINT NUMBER CONTAINS A BINARY EXPONENT.
273 * THE EXPONENT CONTAINS A BIAS OF 128 (200 OCTAL).
274 *
275 * THUS AN EXPONENT OF 0 IS ENTERED AS 200Q, AN EXPONENT OF -10
276 * IS CODED AS 166Q, ETC. THE NUMBER ZERO IS TREATED AS A
277 * SPECIAL CASE: ITS EXPONENT (AND MANTISSA) IS ALWAYS 0.
278 *
279 *
280 * MANTISSA:
281 * -----
282 *
283 * THE MANTISSA OCCUPYS 3 BYTES, FOR A TOTAL OF 24 BITS. THE NUMBERS
284 * ARE STORED IN TWO'S COMPLEMENT NOTATION. THE HIGH ORDER
285 * BIT IS THE SIGN BIT, THE NEXT BIT (100Q) IS THE MOST SIGNIFICANT
286 * DATA BIT. NOTE THAT FLOATING POINT NUMBERS SHOULD ALWAYS
287 * BE NORMALIZED, SO THAT THE MOST SIGNIFICANT DATA BIT IS THE
288 * OPPOSITE OF THE SIGN BIT'S VALUE.
289 *
290 * THE FLOATING POINT ROUTINES SUPPLIED WILL NOT OPERATE ON NON-
291 * NORMALIZED DATA.
292 *
293 * EXAMPLES:
294 * -----
295 *
296 * DECIMAL NUMBER M3 M2 M1 EX
297 *
298 * 1.0 000 000 100 201
299 * 0.5 000 000 100 200
300 * 0.25 000 000 100 177
301 * 10.0 000 000 120 204
302 * 0. 000 000 000 000
303 * 0.1 146 146 146 175
304 * -1.0 000 000 200 200
305 * -0.5 000 000 200 177
306 * -10.0 000 000 260 204

```

## Appendix C

BASIC - WINTER BASIC INTERPRETER.  
FLOATING POINT ROUTINES.HEATH X8ASH V1.0 02/18/77  
13:12:46 01-APR-77 PAGE 7

```
309 ** FPADD - FLOATING POINT ADD.
310 *
311 * ACCX = ACCX + (DE)
312 *
313 * ENTRY (DE) = POINTER TO 4 BYTE FP VALUE
314 * EXIT ACCX = RESULT
315 * SUPPLIED VALUE UNCHANGED
316 * USES A,F
317
318
319 FPADD EQU *
```

```
321 ** FPSUB - FLOATING POINT SUBTRACT.
322 *
323 * FPSUB COMPUTES (DE) - ACCX
324 *
325 * ENTRY (DE) = POINTER TO 4 BYTE FP VALUE
326 * EXIT ACCX = RESULT
327 * SUPPLIED VALUE UNCHANGED
328 * USES A,F
329
330
331 FPSUB EQU *
```

```
333 ** FPNRM - FLOATING POINT NORMALIZE.
334 *
335 * FPNRM NORMALIZES THE CONTENTS OF (ACCX).
336 *
337 * ENTRY NONE
338 * EXIT (ACCX) NORMALIZED
339 * USES A,F
340
341
342 FPNRM EQU *
```

```
344 ** FPNEG - FLOATING POINT NEGATE.
345 *
346 * FPNEG NEGATES THE CONTENTS OF ACCX.
347 *
348 * ENTRY NONE
349 * EXIT (ACCX) = -(ACCX)
350 * USES A,F
351
352
353 FPNEG EQU *
```

BASIC - WINTER BASIC INTERPRETER.  
FLOATING POINT ROUTINES.

HEATH X8ASM V1.0 02/18/77  
13:12:47 01-APR-77 PAGE 8

355 \*\* FPMUL - FLOATING POINT MULTIPLY.  
356 \*  
357 \* ENTRY (DE) = ADDRESS OF Y  
358 \* EXIT ACCX = ACCX \* Y  
359 \* USES A,F  
360  
361  
-----  
362 FPMUL EQU \*

364 \*\* ( FPDIV - FLOATING POINT DIVIDE.  
365 \*  
366 \* ACCX = ACCX/Y  
367 \*  
368 \* ENTRY (DE) = POINTER TO Y  
369 \* EXIT (ACCX) = RESULT  
370 \* USES A,F  
371  
372  
-----  
373 FPDIV EQU \*



## Appendix C

BASIC - WINTER BASIC INTERPRETER,  
ASCII/FLOATING CONVERSION ROUTINES.

HEATH X8ASH V1.0 02/18/77  
13:12:48 01-APR-77 PAGE 9

```

376 ** ATF - ASCII TO FLOATING.
377 *
378 * ATF CONVERTS AN ASCII STRING INTO A FLOATING POINT VALUE
379 * IN ACCX.
380 *
381 * SYNTAX
382 *
383 * NNNN [C,NNN] [E [+ -] NN]
384 *
385 * ENTRY (HL) = ADDRESS OF TEXT
386 * EXIT (HL) UPDATED
387 * (ACCX) = VALUE
388 * USES A,F,H,L
389
390
---*---
391 ATF EQU *

```

```

393 ** FTA - FLOATING TO ASCII.
394 *
395 * FTA CONVERTS A FLOATING POINT NUMBER INTO AN ASCII
396 * REPRESENTATION.
397 *
398 * ENTRY (ACCX) = VALUE
399 * (HL) = ADDRESS TO STORE TEXT
400 * EXIT (A) = LENGTH OF STRING DECODED
401 * (DE) = ADDRESS OF LAST BYTE
402 * USES A,F,D,E
403
404
---*---
405 FTA EQU *

```

BASIC - WINTER BASIC INTERPRETER.  
 FLOATING POINT CONSTANTS.

HEATH XBASH V1.0 02/18/77  
 13:12:49 01-APR-77 PAGE 10

```

408 ** FLOATING POINT VALUES.
409 *
410
411 LON G LIST GENERATED BYTES
412
---*--- 000 000 100 413 FP1.0 DB 0,0,100Q,201Q
201
---*--- 000 000 120 414 FP10. DB 0,0,120Q,204Q
204
---*--- 146 146 146 415 FP0.1 DB 146Q,146Q,146Q,175Q
175
---*--- 000 000 000 416 FP0.0 DB 0,0,0,0
000
---*--- 022 170 233 417 NPI.2 DB 022Q,170Q,233Q,201Q -PI/2
201
---*--- 022 170 233 418 NPI2 DB 022Q,170Q,233Q,203Q -PI*2
203
---*--- 022 170 233 419 NPI DB 022Q,170Q,233Q,202Q -PI
202
---*--- 022 170 233 420 NPI.4 DB 022Q,170Q,233Q,200Q -PI/4
200
---*--- 356 207 144 421 PI.4 DB 356Q,207Q,144Q,200Q PI/4
200
422
423
425 END

```

ASSEMBLY COMPLETE  
 425 STATEMENTS  
 0 ERRORS  
 20312 BYTES FREE

Appendix C



## **APPENDIX D**

### *Entry Points to BASIC Utility Routines*

## Appendix C

## ADDRESSES FOR

EXTENDED BENTON HARBOR BASIC  
ISSUE # 10.01.

|        |         |        |         |        |         |        |         |
|--------|---------|--------|---------|--------|---------|--------|---------|
| ACCX   | 040.066 | ACCY   | 040.074 | ATF    | 076.051 | CVX    | 067.041 |
| CXV    | 067.061 | CXY    | 067.056 | ERR.CB | 064.246 | ERR.CC | 064.236 |
| ERR.DO | 064.267 | ERR.DE | 064.256 | ERR.IC | 065.055 | ERR.IN | 064.274 |
| ERR.IU | 064.302 | ERR.ND | 065.046 | ERR.NV | 064.310 | ERR.OV | 064.317 |
| ERR.RE | 064.330 | ERR.SC | 065.033 | ERR.SL | 064.336 | ERR.SN | 064.350 |
| ERR.SR | 065.016 | ERR.SY | 064.361 | ERR.TC | 064.367 | ERR.TO | 065.004 |
| ERR.TP | 065.077 | ERR.UD | 065.070 | FP0.0  | 103.206 | FP0.1  | 103.202 |
| FP1.0  | 103.172 | FP10.  | 103.176 | FPADD  | 073.106 | FPDIV  | 075.006 |
| FPML   | 074.051 | FPNEG  | 074.030 | FPNRM  | 073.330 | FPSUB  | 073.314 |
| FTA    | 076.336 | IFIX   | 067.226 | IFLT   | 067.264 | NPI    | 103.222 |
| NPI.2  | 103.212 | NPI.4  | 103.226 | NPI2   | 103.216 | PI.4   | 103.232 |
| TUJ    | 071.307 | USRECN | 103.163 | XCY    | 071.326 | ZRQ    | 071.360 |

ADDRESSES FOR  
BENTON HARBOR BASIC  
ISSUE # 05.01.

|        |         |        |         |        |         |        |         |
|--------|---------|--------|---------|--------|---------|--------|---------|
| ACCX   | 040.066 | ACCY   | 040.074 | ATF    | 065.144 | CVX    | 060.171 |
| CXY    | 060.211 | CXY    | 060.206 | ERR.CR | -none-  | ERR.CC | 057.100 |
| ERR.DO | 057.110 | ERR.DE | 057.105 | ERR.IC | 057.154 | ERR.IN | 057.113 |
| ERR.IU | 057.116 | ERR.ND | 057.151 | ERR.NV | 057.121 | ERR.OV | 057.124 |
| ERR.RE | 057.127 | ERR.SC | 057.146 | ERR.SL | -none-  | ERR.SN | 057.132 |
| ERR.SR | 057.143 | ERR.SY | 057.135 | ERR.TC | -none-  | ERR.TO | 057.140 |
| ERR.TF | 057.162 | ERR.UD | 057.157 | FP0.0  | -none-  | FP0.1  | 072.101 |
| FP1.0  | 072.071 | FP10.  | 072.075 | FPADD  | 063.033 | FPDIV  | 064.060 |
| FPMUL  | 063.274 | FPNEG  | 063.260 | FPNRM  | 063.207 | FPSUB  | 063.173 |
| FTA    | 066.006 | IFIX   | 060.373 | IFLT   | 061.031 | NFI    | -none-  |
| NFI.2  | 072.105 | NFI.4  | -none-  | NFI2   | 072.111 | PI.4   | -none-  |
| TDI    | 062.327 | USRFCN | 072.065 | XCY    | 062.346 | ZRO    | 063.000 |



HASL #04.01.00

PAGE 1

|         |             |      |       |              |                |
|---------|-------------|------|-------|--------------|----------------|
| 117.220 |             |      | ORG   | 120000A-160Q |                |
| 063.207 |             |      | EQU   | 063207A      |                |
| 117.220 | 003         |      | START | INX          | B INC UP       |
| 117.221 | 003         |      |       | INX          | B TO           |
| 117.222 | 003         |      |       | INX          | B EXPONENT     |
| 117.223 | 012         |      | LDAX  | B            | (A) = ACCX EXP |
| 117.224 | 247         |      | ANA   | A            | SET CONDX CODE |
| 117.225 | 310         |      | RZ    |              |                |
| 117.226 | 075         |      | DCR   | A            | /2             |
| 117.227 | 312 233 077 |      | JZ    | USR1         | IF UNDER FLOW  |
| 117.232 | 075         |      | DCR   | A            | /2 AGAIN (/4)  |
| 117.233 | 002         | USR1 | STAX  | B            | RET TO ACCX    |
| 117.234 | 315 207 063 |      | CALL  | FPNRM        | NORMALIZE      |
| 117.237 | 311         |      | RET   |              | IN CASE 0      |
| 117.240 |             |      | END   | START        |                |

STATEMENTS = 00016  
FREE BYTES - 10331  
NO ERRORS DETECTED.

## APPENDIX E

### *An Example of USR*

The following program demonstrates BASIC's USR function. This USR program, which performs the simple task of dividing the USR argument by four, was written under TED-8 (see Page 3-37) and assembled by HASL-8 (see Page 4-53). The object program can be written onto tape and then loaded into H8 memory, it can be written directly into memory by HASL-8, or because it is short it can be entered via the H8 keypad. The program works as follows, and is printed out on Page 5-96.

1. When the USR function is called, it finds the starting address of the user-written program at USRFCN. Therefore, the starting address of this program (117 220) is entered in these locations. See "Appendix D" for the Address of USRFCN.
2. The BC register pair point to BASIC's floating point accumulator (ACCX). See Page 5-95 for ACCX description.
3. To divide the number in ACCX by four, the ACCX exponent is shifted left twice. This is accomplished in the following steps.
4. The BC pair is incremented three times, after which it points to the ACCX exponent.
5. The contents of the ACCX exponent is loaded into the 8080 accumulator.
6. The 8080 accumulator is shifted left one, dividing the exponent (and therefore the number) by two.
7. After the first left shift, we test for an underflow, which would indicate the number is out of BASIC's range.
8. If there is no under flow, we shift left again to complete the divide by four.

9. Once the divide by four is complete, the value in the accumulator is placed back in ACCX exponent.
10. We now call the floating point normalization routine (FPNRM) from BASIC. This normalizes the number in the ACCX in case the result was zero.
11. Once normalized, a RET (return) instruction is used to return to BASIC, ending the USR program.

To run this program, you must reserve room at the top of memory for it, and for PAM-8's stack should an RST be executed. For this reason, the program was started 160 (octal) bytes below the top of memory. BASIC must be configured with high memory set below 117 220 (location 20304 decimal). NOTE: You will need at least 12K of memory to use the USR Function.



## INDEX

NOTE: Numbers printed in a bold type face refer to examples of the indicated statement or function.

- ASCII Function, 17-69
- Absolute Value, 17-61
- Addition, 17-14, 17-16
- AND, 17-19
- Arc Tangent Function, 17-62
- Arithmetic, 17-9
- Arithmetic, Functions, 17-61 ff,
- Arithmetic Operators, 17-14
- Arithmetic Priority, 17-14
- Arrays, 17-12 ff, 17-21, 17-33, 17-36
- Assignment Statement, 17-11, 17-45
- Asterisk, 17-7, 17-14
  
- Backspace, changing of, 17-71 (0-20)
- BASIC File, 0-15 ff, 17-30
- Basic Statements, 17-25
- BENTON HARBOR BASIC, 17-7
- Blanks (spaces), 17-73
- Boolean Values, 17-10
- Brackets, 17-26
- BUILD, 17-27
  
- Character Function, 17-68
- CHR \$, 17-68
- CNTRL-H, 17-71
- CNTRL, 17-34 ff, 17-10, 17-52
- CNTRL-Q, 17-72
- CNTRL-S, 17-72
- Checksum Error, 17-30, 17-32
- CLEAR, 17-33, 17-56
- Clear Varname, 17-33
- Clock, 17-36
  - Pause, 17-49
- Colon 17-25, **17-37**, 17-44
- Comma, 17-50 ff,
- Command Completion, 17-7, 17-72
- Command Mode, 17-23 ff, 17-33
- Comments, 17-55
- Concatenation, 17-22
- Continue, 17-23, 17-27, 17-60, 17-58, 17-71
- Control-B, 17-34
- Control-C, 17-71
  - Abort List, 17-47
- Control-O, 17-34, 17-47
  - Abort List, 17-47
- Cosine Function, 17-62
- Cube, **17-34**

- DATA, 17-54 ff, 17-56, 17-54
- Data Exhausted, 17-77
- Data Only Statement,
  - One Line, 17-55
- Decimal Notation, 17-10
- DEF FN 17-58
- DELETE, 17-28, **17-28**
- DIM (Dimension), 17-12, 17-13, 17-36
- Discard Flag CNTRL-O, 17-72
- Discard Flag CNTRL-P, 17-72
- Displays Control, 17-35
- Divide by Zero, 17-75
- Division, 17-14
- Dollar Sign (\$), 17-21
- Double Commas, 17-52 ff,
- DUMP, 17-28
- END, 17-58
- Equal Sign, 17-18, 17-22, 17-46
- Errors, 17-75 ff, 17-42
- Errors Recover, 17-75
- ERROR Table, 17-77
- Exponential Format, 17-9
- Exponential Function, 17-62
- Exponential Notation, 17-9, 17-35
- Exponentiation, 17-14 ff,
- Expressions, 17-14
- Extended B.H. Basic, 17-7
- False, 17-18
- FOR, **17-24, 17-37, 17-39**, 17-39 ff.
- FREE EX, 17-41
- Free Space Function (FRE), 17-67
- Functions, Predefined, 17-61 ff,
- GOSUB, 17-34, **17-41**, 17-42 ff,
- GOTO, 17-44
- High Memory, 17-6
- iexp, 17-26
- IF GOTO, 17-45
- IF THEN, 17-18, 17-45
- IGNORE, 17-30, 17-32
- Immediate Execution, 17-23
- Input and Line Input, 17-59
- Inputting Control, 17-71
- Integer Function, 17-62
- Integer Numbers, 17-9
- I/O MAP, 0-49
- Label File, 0-12 ff,
- Left String Function, 17-69
- LET, 17-46
- Lexical Rules, 17-73
- Line Deletion, 17-74
- LINE Input, 17-59
- Line Insertion, 17-73
- Line Length, 17-74
- Line Numbers, 17-25
- Line Replacement, 17-74
- LIST, 17-47, **17-73**
- LOAD, 17-29
- Loading Basic, 17-7
- Logarithm Function, 17-62
- Loop, 17-39 ff,
- Map, I/O, 17-49
- Map, MEMORY, 17-50
- Maximum Function, 17-67
- Memory, 17-6
  - Poke, 17-49
  - Map, 0-50
- Middle String Function, 17-70
- Minimum Function, 17-67
- Multiple Statements, 17-24
- Multiplication, 17-14 ff,

- "Name", 17-26
- Negation, 17-14, 17-15
- Negotiation, 17-14
- Nesting Depth, 17-40, 17-44
- Nesting, 17-40 ff
- nexp, 17-26 ff
- NEXT, **17-24, 17-37, 17-39, 17-37** ff
- NOT, 17-14, 17-19
- Numeric Data, 17-9
- Numeric Value Function, 17-70
- NXT, 17-57
  
- ON ... GOSUB, 17-48
- ON ... GOTO, 17-48
- Operators, 17-14
- OR, 17-19
- OUT, 17-48
- Output Port, 17-48
- Output Restoration, 17-72
- Output Suspension, 17-72
- Outputting Control, 17-72
  
- PAD Function, 17-63
- Parenthesis, 17-15
- PAUSE, 17-49
- PEEK, 17-63
- POKE, 17-49
- PORT, 17-50
- Position Function, 17-63
- Predefined Functions, 17-61 ff,
- PRINT, **17-37, 17-38, 17-50** ff,
- Printing Strings, 17-51
- Printing Variables, 17-50
- Print Zone, 17-35
- Priority, Arithmetic, 17-14 ff,
- Program Loop, 17-37
- Program Only Mode, 17-25 ff, 17-33, 17-58
- Prompt,
  - Basic, 17-7, 17-71
  - Input, 17-59
  
- Quotes,
  - Input, 17-59
  - Line Input, 17-59
  - Strings, 17-68
  
- Random Function, 17-63
- READ, 17-54 ff,
- Real Numbers, 17-9
- Record, 0-12 ff,
- Relational Operators, 17-18, 17-22
- REM (Remark), **17-55**
- RESTORE, 17-56
- RETURN, 17-35, 17-42 ff,
- Right String Function 17-69
- RND, 17-63
- Rubout, changing of, 17-71 (0-20)
- RUN, 17-30
  
- SCRATCH, 17-31
- Segment Function, 17-65
- Semicolon, 17-52 ff,
- Sep, 17-26
- Sequence Error, 17-29, 17-32
- sexp, 17-26
- SGN, 17-66
- Sign Function, 17-66
- Sine Function, 17-66
- Single Statements, 17-24
- Single Step Execution, 17-57
- Space Function, 17-68
- Spaces, see "Blanks", 17-73
- Special Feature Functions, 17-61 ff,
- SQUARE (Example), 17-24, **17-34, 17-56**
- Square Root Function, 17-66
- Statement Length, 17-25
- Statements, 17-24 ff,
- Statement Types, 17-25
- Step, FOR/NEXT, 17-37 ff,
- STEP, 17-57



- STOP, 17-60
- String Buffers, 17-29
- String Data, 17-10
- String Functions, 17-68
- String Operators, 17-22
- Strings, 17-21
- String Variables, 17-21
- Subroutines, 17-42 ff,
- Subscripted Variables, 17-12
- Subtraction, 17-14, 17-16
- SURE, 17-31
- TAB Function, 17-68
- Tangent Function, 17-67
- Tape Error, 17-30
- Text Rules, 17-73
- Trailing Blanks, 17-24
- True, 17-18
- Truncation, 17-10
- Unary Operators, 17-14 ff,
- USE Error, 17-26
- User Defined Function,
  - Single Line (DEF-FN), 17-56
  - Machine Language (USR), 17-33, 17-66
- VAL 17-70
- VAR, 17-26
- Variables, 17-11, 17-33
- Verify, 17-31

# INDEX

The entries in this Index apply to the introductory section of this Basic Programming Course and to Segments 1 through-14.

'\*' symbol explained, 1-12  
'', use of in PRINT, 3-8  
'/' symbol explained, 1-13  
';', use of in PRINT, 3-6  
ABS function, 5-15  
AND Boolean Operator, 9-14  
ASC function, 8-14  
ASCII number code Chart, 8-9  
ASCII, meaning of, 8-8  
ATN function, 5-11  
Absolute value function, 5-15  
Access to computer memory, 5-21  
Acronym, BASIC, 11  
American Standard Code for Information Interchange, 8-8  
Angular measurement, 5-11  
Anti-Watson, 4-16  
Antilogarithm, 5-13  
Argument, explained, 1-4  
Argument, dummy, in User defined function, 5-25  
Argument, example of, 1-5  
Argument, expression as, 1-8  
Argument, function, use of parentheses, 4-18  
Argument, real, in User defined function, 5-25  
Argument, string, 8-4  
Argument, subscript, 7-11  
Array dimensions, list of, 7-13  
Array of strings, 8-12  
Array, dimensioning two-dimensional, 7-11  
Array, dimensioning, 7-7  
Array, one-dimensional, 7-7  
Array, two-dimensional, 7-10  
Assigning loop value using variable name, 6-10  
Assignment of line numbers, 1-5  
Assignment to variable, 1-10  
Automatically switching to scientific notation, 9-8  
  
BASIC, acronym, 11  
BASIC, development of, 11  
BOOLEAN OPERATOR DEMONSTRATION program, W-58  
Base e, 5-13  
Binary representation, 9-13  
Blank line PRINTed, 1-8  
Body of loop, 6-5



Boolean Operator AND, 9-14  
 Boolean Operator NOT, 9-15  
 Boolean Operator OR, 9-15  
 Boolean Operators, 9-12  
 Boolean values, 9-12  
 Build a Doghouse, Segment 4, 4-1  
 Built-in sub-programs, 5-8  
  
 CHECK SCIENTIFIC NOTATION program, 9-11  
 CHR\$ function, 8-14  
 CLEAR, 9-18  
 CONTINUE, 9-20  
 COS function, 5-11  
 CRT, typical, 1-6  
 CTRL keyboard key, 1-8  
 CTRL/C explained, 1-8  
 CTRL/C, use of to "un-stick" program, 5-7  
 Calculator, difference from computer, 2-2  
 Calling User defined function from program, 5-27  
 Calling a function, method of, 5-10  
 Chart of precedence of operations, 9-17  
 Chart, ASCII number code, 8-9  
 Circular system of angular measurement, 5-11  
 Code, control, 8-8  
 Code, number, ASCII, Chart, 8-9  
 Code, number, typical use of, 8-15  
 Colon, use of for multiple Statements per line, 9-6  
 Column, PRINTed, 3-8  
 Column, as used in two-dimensional array, 7-10  
 Column, width of in PRINT, 3-8  
 Comma, use of in PRINT, 3-8  
 Comma, use of in PRINTing strings, 8-6  
 Comma, use of to separate DATA items, 3-9  
 Comma, use of to separate subscript arguments, 7-11  
 Command mode, W-8  
 Common logarithms, 5-13  
 Computed expressions, 1-8  
 Computer memory, access to, 5-21  
 Computer port, direct input of data from, 5-22  
 Computer port, direct output of data to, 5-22  
 Computer, difference from calculator, 2-2  
 Concatenate, 8-7  
 Concatenation of strings, 8-7  
 Content of variable, use of, 1-11

Content, current, of string variable, 8-6  
Control code, 8-8  
Control portion of loop, 6-5  
Conversations with a Computer, Segment 1, 1-1  
Conversion of BASIC programs, 9-23  
Conversion, common to natural logarithms, 5-13  
Conversion, degrees to radians, 5-11  
Conversion, natural to common logarithms, 5-13  
Conversion, radians to degrees, 5-11  
Conversion, to positive value, 5-15  
Course material explained, 12  
Current content of string variable, 8-6  
Current content of variable, use of, 1-11

DATA list, resetting pointer, 4-17  
DATA, list exhausted, 3-12  
DATA, use of with multiple Statements per line, 9-8  
DATA, used more than once, 3-11  
DATA, 3-9  
DEF, 5-24  
DEMONSTRATION OF LOGARITHMIC FUNCTIONS program, 5-13  
DEMONSTRATION OF SQR function program, 5-9  
DIM, 7-7  
Data item, entry of more than one from keyboard, 3-17  
Data items, INPUT of less than called for, 3-20  
Data, INPUT of string, 8-13  
Data, LINE INPUT of string, 8-13  
Data, direct input of from computer port, 5-22  
Data, direct keyboard entry, 3-16  
Data, direct output of data to computer port, 5-22  
Data, list of, 3-9  
Data, provided when RUNning program, 3-9  
Decimal number code Chart, ASCII, 8-9  
Decimal point, moving of in scientific notation, 9-9  
Decisions, Decisions, Segment 2, 2-1  
Default value, 6-7  
Defining User functions, 5-24  
Definition of program objectives, precise, 4-4  
Degrees, conversion to radians, 5-11  
Detailed plan for writing program, 4-4  
Dimensioning arrays, 7-7  
Dimensioning two-dimensional arrays, 7-11  
Dimensions, array, list of, 7-13  
Direct entry of data from keyboard, 3-16

- Direct input of data from computer port, 5-22
- Direct mode, W-8
- Direct output of data to computer port, 5-22
- Direction to move decimal point in scientific notation, 9-9
- Division symbol explained, 1-13
- Documenting program, use of REM, 3-14
- Dollar sign, use of to mark string variable, 8-5
- Dummy argument in User defined function, 5-25
  
- END, 9-19
- EXAMPLE OF USER DEFINED FUNCTION program, 5-27
- EXP function, 5-13
- EXTRACT SQUARE ROOTS program, 5-6
- E, 9-9
- Element of list, 7-7
- Element of one-dimensional array, 7-7
- Entry of data from keyboard, 3-16
- Entry of more than one data item from keyboard, 3-17
- Equal sign, meaning of in assignment, 1-10
- Error, DATA list exhausted, 3-12
- Error, Syntax, 12
- Example Frame, 13
- Execution pointer, 9-21
- Execution, program, alteration of, 2-4
- Execution, program, order of, 1-5
- Exit, premature, of loop, 6-12
- Exiting a loop, 6-6
- Explaining program, use of REM, 3-14
- Exponent of ten, 9-9
- Exponent, sign of in scientific notation, 9-10
- Exponentiation, 9-18
- Expression as argument, 1-8
- Expression, computed, 1-8
- Expression, consists of, 1-12
- Expression, use in dimensioning array, 7-13
- Expression, use in loop value, 6-10
- Expression, use of variable name in, 1-13
- Expression, 1-8
- Extracting characters from string, 8-16
  
- False, result of Relational test, 2-6
- Flow-Chart example, 4-7
- Flow-Chart symbols explained, 4-5
- Foreword, 8



FOR....NEXT, 6-7  
Frame, example, 13  
Frame, explained, 12  
Function argument, use of parentheses, 4-18  
Function, ABS, 5-15  
Function, ASC, 8-14  
Function, CHR\$, 8-14  
Function, EXP, 5-13  
Function, INT, 5-17  
Function, LEFT\$, 8-16  
Function, LEN, 8-19  
Function, LOG, 5-13  
Function, MAX, 5-20  
Function, MID\$, 8-17  
Function, MIN, 5-20  
Function, PEEK, 5-21  
Function, PIN, 5-22  
Function, POS, 5-23  
Function, RIGHT\$, 8-16  
Function, RND, 5-18  
Function, SGN, 5-16  
Function, SPC, 8-21  
Function, STR\$, 8-18  
Function, TAB, 4-18  
Function, User defined, method of calling from program, 5-27  
Function, User defined, 5-24  
Function, VAL, 8-18  
Function, absolute value, 5-15  
Function, change to ASCII, 8-14  
Function, change to number code, 8-14  
Function, convert numeric argument to string, 8-18  
Function, convert string argument to numeric value, 8-18  
Function, direct input of data from computer port, 5-22  
Function, intrinsic, 5-8  
Function, leftmost characters of string, 8-16  
Function, length of string, 8-19  
Function, maximum number, 5-20  
Function, method of calling, 5-10  
Function, middle characters of string, 8-17  
Function, minimum number, 5-20  
Function, position on line, 5-23  
Function, rightmost characters of string, 8-16  
Function, spaces, 8-21  
Function, types of, 5-8

Function, user, 5-8  
 Functions, logarithmic, 5-13  
 Functions, string, 8-14  
 Functions, trigonometric, 5-11  
  
 GOSUB, target for, 9-7  
 GOSUB....RETURN, 5-4  
 GOTO, optional use of, 9-6  
 GOTO, target for, 9-7  
 GOTO, use of for program jump within loop, 6-13  
 GOTO, 2-4  
  
 How Functions Function, Segment 5, 5-1  
 How to answer questions in Frames, 13  
  
 IF....THEN, 2-9  
 IF, 2-9  
 INPUT TWO DATA ITEMS FROM KEYBOARD program, 3-17  
 INPUT, LINE of string data, 8-13  
 INPUT, entry of more than one data item, 3-17  
 INPUT, of less data items than called for, 3-20  
 INPUT, of string data, 8-13  
 INPUT, use of prompt message, 3-18  
 INPUT, 3-16  
 INT function, 5-17  
 Illegal names for variables, 1-10  
 Initial value of loop, acceptable, 6-9  
 Initial value of loop, 6-5  
 Initial value of variable, 1-11  
 Instruction, Programmed, explained, 12  
 Instruction, complete, 1-4  
 Instructions, program is list of, 1-4  
 Integer function, 5-17  
 Integer subscript, 7-13  
 Integer, 5-17  
 Interlaced loops, 6-12  
 Intrinsic function, 5-8  
 Introduction, Segment 1, 1-2  
 Introduction, 11  
 Isolating characters from string, 8-16  
 Items required to write program, 4-4  
 Items, data, INPUT of less than called for, 3-20

Joining strings, 8-7  
Jump, program, within loop, 6-13  
  
Kemeny, John, 11  
Keyboard, direct entry of data, 3-16  
Keyboard, entry of more than one data item from, 3-17  
Keys, special keyboard, 1-6  
Keyword explained, 1-4  
Keyword, example of, 1-5  
Knowledge of Statements for program writing, 4-4  
Kurtz, Thomas, 11  
  
LEFT\$ function, 8-16  
LEN function, 8-19  
LET, optional use of, 9-5  
LET, 1-10  
LINE INPUT, of string data, 8-13  
LOG function, 5-13  
LOOP DEMONSTRATION program, 6-4  
Leftmost characters of string function, 8-16  
Legal names for variables, 1-10  
Legal variable names, number of, 7-2  
Length of string function, 8-19  
Length of string, 8-4  
Length, term applied to string, 8-11  
Less data items than called for, INPUT of, 3-20  
Line number explained, 1-5  
Line numbers, assignment of, 1-5  
Line numbers, range of, 1-6  
Line, blank, PRINTed, 1-8  
Line, multiple Statements per, 9-6  
Line, position on function, 5-23  
List, as name for one-dimensional array, 7-7  
List, element of, 7-7  
List, of arguments, position in, 2-13  
List, of array dimensions, 7-13  
List, of data, 3-9  
List, of items to be PRINTed, 3-5  
List, of strings, 8-12  
Lists and Arrays, Segment 7, 7-1  
Literal, string, 1-8  
Location, storage, named, term for, 1-9  
Location, storage, named, 1-9  
Logarithms, 5-13



Logical end of program, 9-19  
Logical operation, 9-13  
Loop, Initial value, 6-5  
Loop, Step value, 6-5  
Loop, Terminal value, 6-5  
Loop, body of, 6-5  
Loop, control portion, 6-5  
Loop, exiting, 6-6  
Loop, program jump within, 6-13  
Loop, program, 6-4  
Loops, interlaced, 6-12  
Loops, nested, 6-11  
Loopy de Loop, Segment 6, 6-1

MAX function, 5-20  
MID\$ function, 8-17  
MIN function, 5-20  
Material, Course, explained, 12  
Matrix, as name for two-dimensional array, 7-10  
Maximum number function, 5-20  
Measurement, angular, 5-11  
Memory, computer, access to, 5-21  
Message, error, NO DATA, 3-12  
Message, prompt, for INPUT, 3-18  
Middle characters of string function, 8-17  
Minimum number function, 5-20  
Mode, command, W-8  
Mode, direct, W-8  
Moving decimal point in scientific notation, 9-9  
Multiple Statements per line, 9-6  
Multiple Statements, use of with DATA, 9-8  
Multiple Statements, use of with Relational test, 9-7  
Multiplication symbol explained, 1-12

NAME THE STATE CAPITALS program, W-45  
NATURAL TRIGONOMETRIC FUNCTIONS program, 5-11  
NEXT, 6-7  
NOT Boolean Operator, 9-15  
Name for User defined function, 5-24  
Name, variable, use in assigning loop value, 6-10  
Name, variable, use in dimensioning array, 7-13  
Name, variable, use of in Relational test, 2-11  
Name, variable, used in expression, 1-13

Named storage location, term for, 1-9  
Named storage location, 1-9  
Names, variable, number of legal, 7-2  
Names, variables, illegal, 1-10  
Names, variables, legal, 1-10  
Naming string variables, 8-5  
Natural system of angular measurement, 5-11  
Natural system of logarithms, 5-13  
Negative subscript, 7-13  
Nested loops, 6-11  
Notation, Scientific, 9-8  
Number code Chart, ASCII, 8-9  
Number code, typical use of, 8-15  
Number of legal variable names, 7-2  
Number, line, explained, 1-5  
Number, pseudo-random, 5-18  
Number, random, 5-18  
Numbers, Please!, Segment 3, 3-1  
Numbers, included in string, 8-4  
Numbers, line, assignment of, 1-5  
Numbers, line, range of, 1-6  
Numeric value, conversion from string, 8-18

ON....GOTO, 2-13  
OR Boolean Operator, 9-15  
OUT, 5-22  
Objectives, Segment 1, 1-3  
Objectives, Segment 2, 2-3  
Objectives, Segment 3, 3-3  
Objectives, Segment 4, 4-3  
Objectives, Segment 5, 5-3  
Objectives, Segment 6, 6-3  
Objectives, Segment 7, 7-3  
Objectives, Segment 8, 8-2  
Objectives, Segment 9, 9-3  
One-dimensional array, element of, 7-7  
One-dimensional array, of strings, 8-12  
One-dimensional array, 7-7  
Operation, logical, 9-13  
Operators, Boolean, 9-12  
Operators, Relational, 2-6  
Optional use of GOTO, 9-6  
Optional use of LET, 9-5  
Order of program execution, alteration of, 2-4  
Order of program execution, 1-5

PARALLEL RESISTOR CALCULATOR program, W-29  
 PEEK function, 5-21  
 PIN function, 5-22  
 POKE, 5-21  
 POS function, 5-23  
 PRINT GOLF SCORES program, 7-9  
 PRINT, basic use of, 1-5  
 PRINT, list of items, 3-5  
 PRINT, use of comma, 3-8  
 PRINT, use of semicolon, 3-6  
 PRINT, width of column, 3-8  
 Parentheses, use of for function argument, 4-18  
 Parentheses, use to direct subcomputation, 1-14  
 Pi measure, 5-11  
 Plan for writing program, detailed, 4-4  
 Point, decimal, moving of in scientific notation, 9-9  
 Pointer, DATA list, resetting, 4-17  
 Pointer, execution, 9-21  
 Port Input of data, 5-22  
 Port output of data, 5-22  
 Port, computer, direct input of data from, 5-22  
 Port, computer, direct output of data to, 5-22  
 Position in list of arguments, 2-13  
 Position on line function, 5-23  
 Precedence of operations, 9-16  
 Precise definition of program objectives, 4-4  
 Preciseness, need for, 11  
 Premature exit of loop, 6-12  
 Printer, typical, 1-6  
 Program execution, alteration of, 2-4  
 Program execution, order of, 1-5  
 Program jump within loop, 6-13  
 Program loop, 6-4  
 Program writing, items required, 4-4  
 Program, defined, 11  
 Program, logical end of, 9-19  
 Program, physical end of, 9-19  
 Program, sub-, explained, 5-4  
 Programmed Instruction explained, 12  
 Programs, sub-, built-in, 5-8  
 Prompt message for INPUT, 3-18  
 Prompt, W-9  
 Pseudo-random numbers, 5-18  
 Pythagorean theorem, 5-25



Questions in Frames, how to answer, 13  
Quotation marks, for LINE INPUT of string data, 8-13  
Quotation marks, use of when INPUTting string data, 8-13

RANDOM NUMBER DEMONSTRATION program, W-23  
READ, used more than once, 3-11  
READ....DATA, 3-9  
REM, 3-14  
RESTORE, 4-17  
RETURN keyboard key, 1-8  
RETURN, 5-4  
RIGHT\$ function, 8-16  
RND function, 5-18  
RUNning a program, W-8  
Radian measure, 5-11  
Radian, use of in angular measurement, 5-11  
Radians, conversion to degrees, 5-11  
Random number function, 5-18  
Range of 16 bit binary representation, 9-14  
Range of line numbers, 1-6  
Real argument in User defined function, 5-25  
Relational Operators, 2-6  
Relational test on strings, 8-10  
Relational test with multiple Statements per line, 9-7  
Relational test, use of variable name in, 2-11  
Relational tests, 2-6  
Remarks, 3-14  
Representation, binary, 9-13  
Requirements for program writing, 4-4  
Reserving space for arrays with DIM, 7-7  
Result of expression computation, 1-10  
Review Test, Segment 1, 1-15  
Review Test, Segment 2, 2-16  
Review Test, Segment 3, 3-21  
Review Test, Segment 4, 4-22  
Review Test, Segment 5, 5-28  
Review Test, Segment 6, 6-16  
Review Test, Segment 7, 7-14  
Review Test, Segment 8, 8-22  
Review Test, Segment 9, 9-25  
Rightmost characters of string function, 8-16  
Row, as used in two-dimensional arrays, 7-10

SCORE AVERAGING WITH KEYBOARD ENTRY program, 3-19

SCORE AVERAGING program, 3-15

SCRATCH, W-16

SGN function, 5-16

SIN function, 5-11

SPC function, 8-21

SQR, DEMONSTRATION OF program, 5-9

STEP, 6-8

STOP, example of Statement, 1-4

STORE GOLF SCORES program, 7-8

STR\$ function, 8-18

SUM INTEGERS program, W-28

Scientific notation, 9-8

Seed, use in random numbers, 5-18

Segment 1, "Conversations with a Computer", 1-1

Segment 1, Objectives, 1-3

Segment 1, Review Test, 1-15

Segment 2, "Decisions, Decisions!", 2-1

Segment 2, Objectives, 2-3

Segment 2, Review Test, 2-16

Segment 3, "Numbers, Please!", 3-1

Segment 3, Objectives, 3-3

Segment 3, Review Test, 3-21

Segment 4, "Build a Doghouse", 4-1

Segment 4, Objectives, 4-3

Segment 4, Review Test, 4-22

Segment 5, "How Functions Function", 5-1

Segment 5, Objectives, 5-3

Segment 5, Review Test, 5-28

Segment 6, "Loopy de Loop", 6-1

Segment 6, Objectives, 6-3

Segment 6, Review Test, 6-16

Segment 7, "Lists and Arrays", 7-1

Segment 7, Objectives, 7-3

Segment 7, Review Test, 7-14

Segment 8, "String Savers", 8-1

Segment 8, Objectives, 8-2

Segment 8, Review Test, 8-22

Segment 9, "Tricks of the Trade", 9-1

Segment 9, Objectives, 9-3

Segment 9, Review Test, 9-25

Semicolon, use of in PRINT, 3-6

Semicolon, use of in PRINTing strings, 8-6

Separation of subscript arguments, 7-11

Sign function, 5-16  
Sign of exponent in scientific notation, 9-10  
Sign, dollar, use of to mark string variable, 8-5  
Sign, equal, meaning of in assignment, 1-10  
Simple variable, 7-2  
Sixteen bit binary representation, range of, 9-14  
Slots in User defined functions, 5-25  
Space for arrays, reserving with DIM, 7-7  
Space reserved with DIM, 7-8  
Spaces function, 8-21  
Special keyboard keys, 1-6  
Stack of Money program, final, 4-20  
Statement, instruction is, 1-4  
Statements, multiple per line, 9-6  
Step value of loop, acceptable, 6-8  
Step value of loop, 6-5  
Stopper, use of in READ....DATA, 3-9  
Stopper, use of more than one, 3-13  
Storage location, named, term for, 1-9  
Storage location, named, 1-9  
String Savers, Segment 8, 8-1  
String argument, 8-4  
String data, INPUT of, 8-13  
String functions, 8-14  
String literal, 1-8  
String variable, current content of, 8-6  
String variable, use of dollar sign to mark, 8-5  
String variable, 8-5  
String variables, naming, 8-5  
String, conversion from numeric value, 8-18  
String, explained, 8-4  
String, extracting characters from, 8-16  
String, isolating characters from, 8-16  
String, length of, 8-4  
String, middle characters of function, 8-17  
String, numbers included in, 8-4  
String, term length applied to, 8-11  
Strings, Relational test on, 8-10  
Strings, concatenation of, 8-7  
Strings, joining, 8-7  
Strings, lists and arrays of, 8-12  
Sub-program, explained, 5-4  
Sub-programs, built-in, 5-8  
Subcomputation, use of parentheses to direct, 1-14



Subscript, integer, 7-13  
Subscript, negative, 7-13  
Subscripted variable, 7-6  
Switching to scientific notation, 9-8  
Symbol for division explained, 1-13  
Symbol for multiplication explained, 1-12  
Symbols, Flow-Chart, explained, 4-5  
Syntax error, 12

TAB function, 4-18  
TAN function, 5-11  
TEMPERATURE CONVERSION program, 9-20  
THEN, 2-9  
TWO DIMENSIONAL ARRAY DEMONSTRATION program, 7-12  
Targets for GOTO and GOSUB, 9-7  
Term for named storage location, 1-9  
Terminal value of loop, acceptable, 6-9  
Terminal value of loop, 6-5  
Test, Relational, on strings, 8-10  
Test, Relational, use of variable name in, 2-11  
Test, Relational, with multiple Statements per line, 9-7  
Test, Relational, 2-6  
Test, Review, Segment 1, 1-15  
Test, Review, Segment 2, 2-16  
Test, Review, Segment 3, 3-21  
Test, Review, Segment 4, 4-22  
Test, Review, Segment 5, 5-28  
Test, Review, Segment 6, 6-16  
Test, Review, Segment 7, 7-14  
Test, Review, Segment 8, 8-22  
Test, Review, Segment 9, 9-25  
Tricks of the Trade, Segment 9, 9-1  
Trigonometric functions, 5-11  
True, result of Relational test, 2-6  
Two-dimensional array, of strings, 8-12  
Two-dimensional arrays, 7-10

Up-arrow, 9-18  
Use of REM to document or explain program, 3-14  
Use of parentheses to direct subcomputation, 1-14  
Use of variable content, 1-11  
Use of variable name in expression, 1-13  
User defined functions, 5-24  
User function, 5-8

VAL function, 8-18  
Value of numbers included in string, 8-4  
Value of variable, initial, 1-11  
Value, Initial, of loop, 6-5  
Value, Step, of loop, 6-5  
Value, Terminal, of loop, 6-5  
Value, absolute, function, 5-15  
Value, default, 6-7  
Value, loop, acceptable types, 6-8  
Value, loop, use of variable name in assigning, 6-10  
Value, numeric, conversion from string, 8-18  
Value, positive, conversion to, 5-15  
Values, Boolean, 9-12  
Variable name, use in assigning loop value, 6-10  
Variable name, use in dimensioning array, 7-13  
Variable name, use of in Relational test, 2-11  
Variable name, used in User defined functions, 5-24  
Variable name, used in expression, 1-13  
Variable names, number of legal, 7-2  
Variable, assignment to, 1-10  
Variable, content, use of, 1-11  
Variable, explained, 1-9  
Variable, illegal names for, 1-10  
Variable, initial value of, 1-11  
Variable, legal names for, 1-10  
Variable, simple, 7-2  
Variable, string, current content of, 8-6  
Variable, string, naming, 8-5  
Variable, string, use of dollar sign to mark, 8-5  
Variable, string, 8-5  
Variable, subscripted, 7-6  
Variables, use of more than one for INPUT, 3-17  
Vocabulary, BASIC, 1-2  
  
Watson, anti-, explained, 4-16  
What's all this about Computer Languages, 9  
Width of column in PRINT, 3-8  
  
Zero, initial value of variable, 1-11